IMPROVING PROCESSOR PERFORMANCE WITH MULTIPLE-ACCESS CACHES AND EARLY-RESOLUTION OF BRANCHES AND LOADS

By

BYUNG-KWON CHUNG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1999

To my families

ACKNOWLEDGMENTS

First of all, I would like to express my thanks to the committee members: Dr. Randy Chow, Dr. Sanguthevar Rajasekaran, Dr. Abdelsalam Helal, and Dr. Paul Chun. Special thanks to Dr. Paul Chun for his support during early days of my graduate study. More importantly, many thanks to my advisor Dr. Jih-Kwon Peir for his support, guidance, and encouragement.

There are countless people who motivated and encouraged me to finish the study. Many of my friends both in Korea and across the nation helped me get through. Specially, I would like to thank Mr. Chang-Shin Park and his wife, and Rev. Heung-Yeol Yoo in Seattle WA who cheered me up in the final stage of my study. Also, for Rev. Hee-Young Sohn and his wife in Gainesville FL who taught me the real goal of the life. There are many local church members and friends who shared with me for both rainy and shiny days of the life in Gainesville.

I appreciate Cho-ke Chung family foundation in Wonju Korea for partial financial support. My families deserve to receive my deep gratitude without a question: father-in-law, mother-in-law, my three bothers, and my sister. I would like to keep the memories of care that my father had shown to me before he past away. I deeply owe my mother Oak-Shim Kim for her encouragement and unconditional love. My

lovely children, Jeffrey and Grace, allowed me to relax by having fun during the stressful days. Lastly, inexplicable thanks go to my wife Young-Ran. It is hard to imagine for me to finish my study without her sacrifice, devotion, care, and love. Thank the Lord.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

IMPROVING PROCESSOR PERFORMANCE WITH MULTIPLE-ACCESS CACHES AND EARLY-RESOLUTION OF BRANCHES AND LOADS

By

Byung-Kwon Chung

December, 1999

Chairman: Dr. Jih-Kwon Peir
Major Department: Computer and Information Science and Engineering

One of the major metrics to determine processor performance is *cycles per instruction* (CPI). *Pipelining* is a classical design technique to improve CPI so that processor performance can improve. Modern processors employ *superscalar* pipelining technique to execute more than one instruction per cycle (IPC) to achieve even higher CPI improvement. The improvement by superscalar pipelining, however, is not linear to the number of execution units in superscalar pipeline due to limited instruction level parallelism (ILP). In designing a high-performance pipelined processor, instruction and data memory accesses play a critical role. Because of the increasing performance gap between processor and memory, small and fast storage in a form of cache memory is used to bridge the performance gap.

A multiple-access cache is a direct-mapped cache that may be accessed more than once, each time with a different hash function, to satisfy a memory request. Better solutions on multiple-access caches are proposed. Those are the rehash functions that randomize the distances in terms of the number of cache sets, the search and replacement based on the reference order, and an indirect access mechanism on the cache data array to eliminate the extra data movement. To overcome limited ILP in processor pipelining, highly accurate prediction-based speculation techniques are proposed by resolving *branch* and *load* instructions ahead of normal program execution. Early-resolution of load and branch instructions can be done by efficiently building a data-flow graph to track the flow of data from producer to consumer dynamically. The early-resolution technique can be able to transcend barriers such as the instruction execution window to trigger a consumer instruction whenever the producer instruction is completed. The early-resolution technique can also be applied solely to load-address prediction with very high accuracy.

CHAPTER 1
INTRODUCTION

There are three factors that can determine processor performance, i.e., *clock cycle*, *cycles per instruction* (CPI), and *instruction count* of certain representative programs. The clock cycle time depends on both hardware technology and organization, and the instruction count depends on both complier technology and instruction set architecture. The CPI, on the other hand, depends on both instruction set architecture and hardware organization [16]. By concentration on hardware organization, advanced architectural techniques to improve clock cycle time and/or CPI of future microprocessors are pursued in this dissertation.

Pipelining is a classical technique used in today's microprocessors to overlap instruction executions, so that faster cycle time and/or lower CPI can be achieved. To further improve the instruction execution rate, *superscalar* pipelining technique is capable of executing more than one instruction per cycle (IPC) by fetching, decoding, and executing more than one instruction at a time. However, due to limited instruction level parallelism (ILP), the improvement of IPC is limited to about four instructions on the average.

Instruction and data memory accesses play a critical role in designing a high-performance pipeline. Due to the increasing performance gap between processor and

memory, small and fast storage in the form of *cache* memory is used to bridge the performance gap. The performance of a cache can be determined by two primary factors—the time needed to access the cache and the fraction of memory references that can be satisfied by the cache. However, these two parameters are normally opposing, i.e., to improve one parameter normally impairs the other parameter.

After investigating the ILP constraints in the processor pipeline and two cache access parameters, efficient solutions to these critical issues are proposed and evaluated in this dissertation.

1.1 Issues in Memory Hierarchy

According to the *principle of locality*, programs access a relatively small portion of their address space at any given period of execution. There are two types of locality: *temporal locality* and *spatial locality*. Temporal locality is a property that a previously referenced item will tend to be referenced soon again. Spatial locality is a tendency that nearby addresses to a referenced item will be referenced soon. A *memory hierarchy* takes advantage of the principle of locality by organizing multiple levels of memory with different sizes and speeds. Caches usually reside in the higher level in the memory hierarchy to have fast access time for a majority of memory references. A performance gap between processor and memory can be reduced if a large portion of memory references is satisfied by the caches.

Usually cache memories are organized as two-dimensional arrays. The first dimension is *set* and the second dimension is *set associativity*. The *set* ID is determined

by a function of the memory request address. The cache *line* ID within a set is decided by comparing the address tags in cache set and the reference address. Caches with set associativity one are referred to as *direct-mapped caches* while caches with set associativity greater than one are referred to as *set-associative caches.* If the number of sets is one, caches are called *fully-associative.* Because a requested reference can reside anywhere in the fully-associative caches, there exist no misses due to cache placement collision, in other words, no *conflict misses.* On the other hand, direct-mapped caches can suffer from conflict misses.

The ability of caches to bridge the performance gap between processor and memory is determined by two primary factors—the time needed to retrieve data from the cache and the fraction of memory references that can be satisfied by the cache, commonly referred as *access (hit) time* and *hit ratio* respectively [33].

A *multiple-access cache* [1] is a direct-mapped cache which is accessed more than once in order to achieve both the access time of a direct-mapped cache and the hit ratio of a set-associative cache. Fast hit time can be achieved by having as many hits as possible on the first cache access, and high hit ratio can be satisfied by accessing a cache multiple times. In multiple-access caches, the proper *search* order and *replacement* algorithm are crucial to achieve both fast hit time and high hit ratio. In addition to this, a *rehash function* to guide next search location needs to be considered because recent studies [32, 13] show that the conventional set-associative caches are not resilient to certain programs which exhibit high conflict misses.

The essential issues related to multiple-access caches are investigated in this dissertation. Based on the investigation, better solutions are proposed as a way of improving processor performance. The new proposals on multiple-access caches are validated by performance simulation.

1.2 Issues in Processor Pipelining

Pipelining overlaps multiple instruction executions by exploiting instruction level parallelism to improve cycle per instruction (CPI) or cycle time of computer systems. However, there exist inherent dependencies in the sequential instruction stream which make the pipelined processor underutilized. These dependencies prevent the subsequent instructions from executing in their designated pipeline stage. The pipeline can be stalled because of *control dependency* if the pipeline cannot proceed due to previous branch or jump instructions. The pipeline can also be stalled because of *data dependency* if the register or memory value of the previous instruction is unknown. Data dependency can be further specified into *load dependency* or *load interlock* if the previous instruction outcome is produced from data memory by a load instruction. Both load and branch instructions are especially critical because they are often at the beginning of dependency chains. To avoid inefficiency due to the dependencies, speculative instruction execution by predicting previous instruction's outcomes, which are the source of the dependency, has emerged. Naturally, the performance of prediction-based speculation largely relies on the prediction accuracy. In speculative processors, performance impact is very sensitive to prediction accuracy

of branch and load instructions because the impact of an incorrect prediction will be higher.

The best performing *branch* prediction schemes, so called *correlation-based* predictors, rely on past branch behaviors. Various accurate branch prediction schemes have been proposed in the literature [42, 26, 25]. However, recent studies also show that the behavior of certain classes of branches is very difficult to predict [11] suggesting the correlation-based predictors almost reached their upper bound. Several schemes for predicting *load* instruction outcome are proposed in the literatures [10, 3, 21, 5]. The prediction accuracies of these schemes, however, are not satisfiable because the memory access pattern of load instructions is not uniform especially in integer-intensive programs.

In this dissertation, a different direction to achieve high prediction accuracy for both branch and load instructions is presented. The rationale of this new approach is to execute the branch and load instructions early and produce the expected results ahead of normal instruction flow. To achieve this, the proposed approach dynamically keeps track of the relationships of producer to consumer by efficiently building a data-flow graph. A few hardware structures are constructed not only to build instruction dependency but also to trigger instruction early. Because knowing the load instruction address in the processor pipeline, instead of load outcome, can also reduce the latency due to load dependency, a new load address prediction scheme is also proposed using the same idea of producer and consumer relation as before.

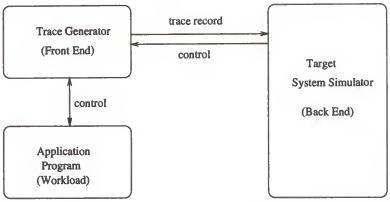


Figure 1.1. On-the-fly trace-driven simulation environment

A simple machine model is constructed and performance simulation is followed to justify the effectiveness of this approach.

1.3 Performance Evaluation Method and Workload

Figure 1.1 shows on-the-fly trace-driven simulation environment for both multiple-access caches and processor pipelining. For cache studies, a tracing and instruction-set simulation tool *Shade* [39] from Sun Microsystems under Solaris is used. Shade controls the application program execution and sends trace records of interest to the target system simulator. The target simulator interacts with Shade by calling Shade libraries. For studies of processor pipelining, a processor simulation tool set *Simplescalar* [6] is used. The interaction of the Simplescalar and the target simulator is similar to Shade. The detailed machine model for each one will be explained in the corresponding chapters.

Benchmark programs are intended to provide a measure to compare performance. The SPEC95 [37] from the Standard Performance Evaluation Corporation is one of the most widely used benchmark programs in our research community.

The SPEC95 consists of two types of benchmarks: one is the SPECint95, a set of integer-intensive benchmarks, and the other is the SPECfp95, a set of floating-point intensive benchmarks. The SPEC95 benchmarks provide comparative measures of performance across a wide range of hardware and the benchmarks are developed from real user applications.

1.4 Outline

The subsequent chapters of this dissertation are organized as follows. In Chapter 2, the essential issues of multiple-access caches, which attempt to achieve high hit ratios while maintaining fast cache access time, are discussed. These issues include mathematical functions for directing multiple-access locations, a search and replacement algorithm to provide a lower cache miss ratio as well as a lower number of searches, and data swapping issue which is claimed to be a major criticism in multiple-access cache design. For each of these issues, better solutions are presented and validated with simulation. Chapter 3 investigates processor pipeline bottlenecks which hamper processor performance, and presents new solutions to overcome the bottlenecks. The bottlenecks are due to control and data dependencies in application programs. With the trend of prediction-based speculative execution in modern pipeline design, new approaches to achieve better prediction accuracy for both control and data access instructions are presented and validated. Applying the idea proposed in the previous chapter, Chapter 4 studies specified address prediction schemes and presents a new address prediction mechanism with the evaluation result. Finally, Chapter 5 concludes the study of this dissertation.

CHAPTER 2
ESSENTIAL ISSUES IN MULTIPLE-ACCESS CACHES

2.1 Introduction

According to the *principle of locality*, programs access a relatively small portion of their address space at any given period of execution. A *memory hierarchy* takes advantage of the principle of locality by implementing multiple levels of memory with different sizes and speeds. Thus, caches are critical components in high-performance processors. The ability of caches to bridge the performance gap between processor and memory is determined by two primary factors—the time needed to retrieve data from the cache and the fraction of memory references that can be satisfied by the cache, commonly known as *access (hit) time* and *hit ratio* respectively [33]. Ideally, the caches with both a high hit ratio and a fast access time would be desirable. But, in conventional caches, these two dimensions are opposing. For example, the absence of the line selection in direct-mapped caches enables a faster access time than that of set-associative caches of the same size. On the other hand, direct-mapped caches tend to have lower hit ratios due to the higher conflict misses when nearby memory requests target different cache lines allocated in the same set [17].

Multiple-access caches try to achieve high hit ratios while maintaining fast access time by accessing the direct-mapped cache more than once for a memory request,

each time with a different hash function [1, 2]. A fast access time is achieved when the requested data are located at the *primary* location by the primary hash function. When the requested line is not found in the primary location, the cache is accessed again by a *secondary* location with the secondary hash function. A small penalty is paid if the data are located at the secondary location. Data swapping between the primary and secondary locations may be needed to increase the hit ratio to the primary location. When a miss occurs, the requested line is always placed in the primary location while the line in the primary location is either evicted or moved to the secondary location depending on the replacement algorithm.

Traditionally, in order to achieve a fast access time, the simple direct-mapped hashing using the low-order bits of the line address, also known as bit-selection mechanism [33], is chosen as the primary hash function. The rehash function of flipping the highest-order index bit is adapted [1] to resemble the 2-way set-associative design.

In this dissertation, the essential issues in multiple-access caches are investigated and better approaches to multiple-access caches are proposed. First, besides flipping the highest-order index bit, more general rehash functions are considered. The initial results show that the more generalized rehash functions provide marked improvement in terms of hit ratios for particular workloads. Second, Least-Recently-Used(LRU) based search and replacement is proposed. The simulation results from trace-driven simulation show that this simple LRU-based search/replacement mechanism can achieve better performance over the conventional multiple-access caches. Lastly, an indirect access mechanism on the cache data array is proposed to eliminate

the extra data movement. A further timing simulation is followed to evaluate the indirect access mechanism.

2.2 Multiple-Access Caches

A *multiple-access cache* [1, 2] is a direct-mapped cache which is accessed more than once in order to achieve the access time of a direct-mapped cache and the hit ratio of a set-associative cache. Fast hit time can be achieved by having as many hits as possible on the first cache access, and high hit ratio can be satisfied by accessing the cache multiple times. An example of a multiple-access cache is shown on the right side of Figure 2.1 where the cache is 8KB, direct-mapped, and a 32B cache line. As can be seen from the figure, the cache structure not only resembles direct-mapped caches to achieve fast hit time but also has two hash functions. So, it shows similar behavior as 2-way set-associative caches which have less conflict misses than direct-mapped caches. When the requested line is not found by the first hash function at the *primary* location, the cache is accessed again by the second hash function or rehash function at the *secondary* location.

A *hash-rehash* multiple-access cache is proposed by Agarwal et al. [1]. It is based on the direct-mapped cache with a simple rehash function of flipping the highest-order index bit. In this design, a fast hit time can be obtained when the requested line is found in the primary location. Upon a hit to a secondary location, the lines located in the primary and secondary locations are swapped. When a cache miss occurs, the line in the primary location is moved to the secondary location, the requested line is fetched into the primary location, and the line in the secondary location is

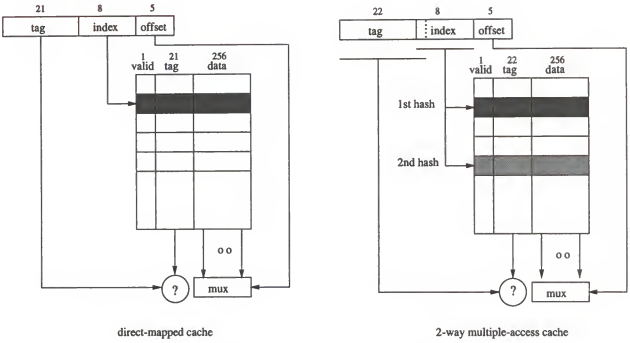


Figure 2.1. Direct-mapped cache and its conceivable 2-way multiple-access cache

evicted. Although this scheme appears similar to a 2-way set-associative cache with the LRU replacement policy, it likely has inferior hit ratio. This is because the primary location for a reference may be the secondary location for another reference and vice versa. In other words, this simple scheme always searches and replaces the line at the secondary location without considering the MRU/LRU order between the two lines.

The *column-associative* cache [2] uses the same rehash function by flipping the highest-order index bit. In order to improve the replacement algorithm, a *rehash bit* in each tag array entry is maintained to indicate that the corresponding line has been accessed using the secondary hash function to guide the search and replacement. When a primary access misses a line with the rehash bit turned on, it is unnecessary to search the secondary location and the line in the primary location is simply replaced.

The rehash bit helps to identify the correct primary to secondary sequence. However, the MRU/LRU order is still missing when both locations are used as a primary location. In this design, these two locations are referred as a *column*.

Recently, *extended column-associative cache* [44] is proposed to allow more than two lines in a column in order to improve the overall cache hit ratio further. In this design, the rehash functions are defined by changing a few high-order index bits. For instance, in a 4-way column-associative cache(details are shown in Figure 2.4 of section 2.4), four direct-mapped cache sets indexed by '00xxxx', '01xxxx', '10xxxx' and '11xxxx' belong to the same column. When any one of the four is accessed as the primary location, the other three are the secondary locations. The access of the extended column-associative cache is straightforward. First, an attempt to access the primary location is made by matching the address tag of the request with the tag stored in the direct-mapped location of the tag array. Second, when the requested data are not found in the primary location, the search proceeds through other secondary locations in the same column. In case the line is found in a secondary location, the requested data are accessed and the requested line is swapped with the line in the primary location to increase the hit to the primary location. Finally, when a miss occurs, the line in the primary location is first moved to the LRU location in the respective column and the requested line fetched from the lower memory hierarchy is placed in the primary location. One critical performance issue in this extended column-associative cache design is to provide an efficient search mechanism when the requested line is not located in the primary location. In this scheme, an extra

hardware of *index vector* for each column is maintained. The index vector indicates the locations of the rehashed lines from each primary location.

Common criticism on the multiple-access caches is that a penalty is paid if the data are located at the secondary location. Data swapping between the primary and secondary locations may be needed to increase the hit ratio to the primary location. Regarding data swapping, an indirect cache access technique using a Most-Recently-Used(MRU) bit array or a steering table has been proposed to avoid data swapping at the expense of a potentially longer cache access delay [20, 7].

In addition to the above issues, recent studies for cache set-selection randomizations [32, 13] suggest that the simple bit-selection *hashing function* of conventional set-associative caches is not resilient especially for floating-point intensive programs which exhibit high conflict misses. Table 2.1 shows some of the results. From these studies, the original rehash function of multiple-access caches can also cause high conflict misses for floating-point intensive programs.

Table 2.1. The miss ratios(%) of SPECfp95 for 2-way 8KB data caches [13]

workload	swim	tomcatv	turb3d	wave5
overall	59.1	48.1	6.5	31.7
conflict	51.2	36.4	3.7	17.8

In summary, based on the above observations, the originally proposed scheme [1] and variations to the original scheme [20, 2, 7, 44] pose the following problems.

- The simple rehash function to the secondary location can cause high conflict misses as shown in Table 2.1.

- The existing multiple-access cache schemes [1, 20, 2, 7, 44] are inefficient in terms of search sequence and replacement algorithm due to lack of MRU/LRU information.

- There exists overhead of data swapping when an access is a hit in the secondary location.

In this dissertation, the essential issues in multiple-access caches are discussed and better solutions to each of these issues are proposed. In the subsequent sections, these issues are explained in detail.

2.3 Rehash Functions

A multiple-access cache based on the direct-mapped cache with a simple rehash function of flipping the highest-order index bit was proposed by Agarwal et al. [1]. This simple design resembles a 2-way set-associative cache, in which each pair of the primary and the secondary locations is placed in the same set indexed by the lower-order bits of the line address so that they can be searched in parallel. It is known, however, set-associative caches may suffer high conflict misses for certain floating-point intensive workloads [13].

In a multiple-access cache, the cache may be accessed more than once for a memory reference, each time with a different hash function. In order to achieve a fast access time, the simple direct-mapped hashing is normally used as the primary hash function. The rehash function, which is used only after the requested data are absent in the primary location, has the flexibility in selecting the index bits. It is convenient, however, to restrict the hash and the rehash as an *inverse* function

to each other, i.e., when the primary location of a group of lines is the secondary location of another group of lines, the primary location of the second group must be the secondary location of the first group. This inverse function allows the lines to be moved between the primary and the secondary locations for improving the hit ratio to the primary location. As a contrary example, for instance, assume B is the secondary location of A, and C is the secondary location of B. In case when both B and C are occupied as the secondary locations, the lines in B and C cannot be swapped upon a hit to the secondary line in C because the line moved out of B will be lost.

In this dissertation, three categories of rehash functions are proposed. The basic idea is to randomize the set distance between primary and secondary locations because it can reduce potential conflict misses between each other.

1. Full or partial index complement scheme: Complement all or partial bits from the direct-mapped indices.

2. Bit-wise exchange scheme: Partition the index bits into two groups and perform bit-wise exchange between the groups, for example, swapping either the high-order and the low-order or the even and the odd index bits.

3. Shuffle/unshuffle scheme: Divide the cache sets into two groups based on the value of the most-significant index bit and use the shuffle/unshuffle interconnection pattern to select partners, one from each group.

There are many combinations within each category of rehash functions. Figure 2.2 illustrates a few interesting ones. The original rehash function of flipping the

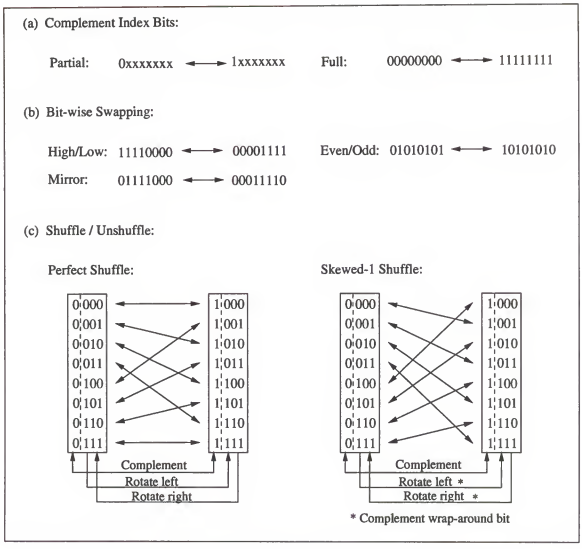


Figure 2.2. Three categories of rehash functions

most-significant index bit can be considered as a special case of the index-complement scheme. As mentioned, this scheme may suffer high conflict misses when the memory addresses of consecutive references differ by a power-of-two stride. One simple cure is to complement all the index bits. Another straight-forward solution to reduce the conflict misses is to rearrange the index bits by performing bit-wise exchange among the direct-mapped indices. Three cases can be considered, i.e., to exchange each high-order bit with its low-order counterpart pairs, to exchange even and odd, and

to exchange bit 0 with bit n-1, bit 1 with bit n-2 and so on from the n-bit indices. Although simple, the bit-wise exchange may encounter identical mapping and reduce its effectiveness. Certainly, a hybrid scheme which applies both the complement and the bit-wise exchange is also a viable approach.

The shuffle/unshuffle scheme uses the perfect shuffle interconnection [38] for matching a pair of primary and secondary locations. A skewed-d perfect-shuffle is also considered. The skewed scheme shifts d sets at the initial mapping as shown in Figure 2.2 for randomizing the distance between each pair of primary and secondary locations. The shuffle/unshuffle rehash function requires only simple bit-wise logical operations. The most-significant index bit is always complemented, and the rest of the index bits are either rotated to the left or to the right depending on the value of the most-significant bit as illustrated in Figure 2.2. In the skewed-1 shuffle scheme, the wrap-around bit needs to be complemented during the rotation. Note that except for complementing partial index bits, the other rehash functions require to include the entire index bits as part of the tags in order to avoid mis-identifying the lines located in the secondary locations.

2.4 Search and Replacement Algorithms

In conventional cache accesses, the memory address can be divided into three disjoint regions: the address tag, the cache index, and the line offset. For column-associative caches based on changing/rehashing of the high-order index bits, those high-order index bits are overlapped with the low-order tag bits as shown in Figure 2.3. In this figure, A_3 represents the offset bits, A_1A_2 (A_1 concatenated by A_2)

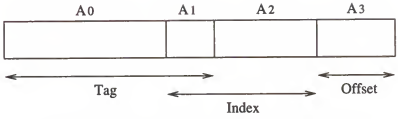
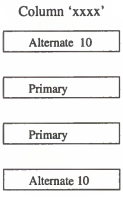


Figure 2.3. Column-associative cache address



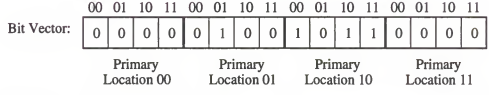


Figure 2.4. 4-way extended column-associative search with index vector

are the index bits, and A_0A_1 are the tag bits. The primary location is indexed by A_1A_2 of the memory address, in which A_1 represents those high-order rehashed index bits. A_1 must be included in the tag in order to identify a hit when the line is located in a secondary location. Therefore, in a cache access, whether the primary location contains a primary line, i.e., the line is accessed using the primary function, can be determined by matching A_1 of the memory address with A_1 of the line located at the primary location. Several observations can be found for the existing column-associative caches as follows.

- The primary location must contain the most-recently-used line among the lines with matching A_1 in the respective column. This is true under the LRU replacement policy and the swapping/placement of the newly requested line to the primary location. Therefore, whenever the primary location contains a line rehashed by another primary location, the search to other secondary locations is unnecessary. For example, in Figure 2.4, if the first hash function of a memory request address targets either location 0 or location 3, a miss occurs and no further search is necessary because they are rehashed by location 2.

- When a request misses the primary location, the search to the secondary locations is necessary if A_1 of the requested address matches that of the primary location tag. However, in this case, only those cache lines that are less-recently-used than the line in the primary location are searched. This LRU sequence reduces the number of searches for misses (this is also true for hits as explained in the following observation). For instance, if the line in the primary location 1 in Figure 2.4 is the LRU line in the column, no further search is required upon a miss to this primary location.

- The search according to the LRU sequence may further help to speedup the hit time to the secondary locations. For example, when a request misses the primary location 2 in Figure 2.4, the search based on the LRU sequence may locate the line in either location 0 or location 3 faster because of the program locality. But, the index-vector scheme does not indicate which location should be probed first between location 0 and location 3. The reason is that the index

vector does not provide the MRU/LRU order among the lines rehashed from the same primary location.

- Since we can know the line is rehashed by another primary location if A_1 does not match, the originally proposed rehash bit is not needed. Although the index-vector scheme eliminates any useless search, it requires a n^2 bit-map for each column with n lines. But, $n^2 - n$ bits are always empty as shown in Figure 2.4.

Based on the above observations, an efficient search algorithm can be proposed in column-associative cache, coined as *LRU-based* column-associative cache, and the algorithm includes the following steps.

1. Search Primary Location

   Upon receiving a memory request, the primary location is always probed first. If the line is found, the requested data are accessed like a regular direct-mapped cache. The LRU sequence is updated after each memory request to reflect the most-recently-used cache line.

2. Determine Whether to Search Alternative Locations

   In case the requested line is not located at the primary location, the search for the line in other secondary locations is continued only when A_1 bits of the requested address match A_1 bits of the line in the primary location.

3. Search Sequence of Alternative Locations

   The search of the line in the secondary locations follows the LRU sequence.

It starts from the next less-recently-used location with respect to the primary location until the least-recently-used location in the respective column. If the line is found in the secondary location, the requested data can be accessed with certain delays due to the additional search. Afterwards, the matched line is swapped with the line located in the primary location, and the LRU information is updated accordingly.

4. Handle Cache Misses

   A cache miss occurs when the requested line is not located in the respective column. In this case, the line in the primary location will be moved to the LRU location and the requested line fetched from the lower-level memory hierarchy will reside in the primary location.

According to the new search algorithm, the access to the original 2-way column-associative cache is straightforward. When a memory request misses the primary location, the search to the secondary location is needed only when the primary location is MRU. This is due to the fact that the MRU line must always be located at its primary location. In comparison with the column-associative cache with rehash bits, this newly proposed scheme will not only achieve a 2-way set-associativity cache hit ratio but also reduce unnecessary searches to the secondary location when both locations are used as a primary location. For associativity of 4 or above, the search to the secondary location is needed either when the primary location is not LRU or when A_1 bits of the primary location match with that of the requested address.

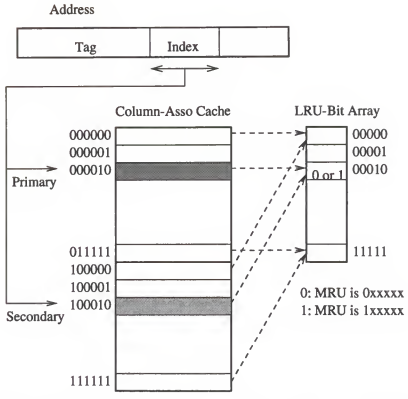


Figure 2.5. Improved column-associative cache using LRU bits

In Figure 2.5, an illustration of an improved 2-way column-associative cache structure is shown. Each bit of the LRU bit array is associated with two locations in each column. When the LRU bit is equal to '0', the MRU line is located in the set 0xxxxx; the MRU line is located in the set 1xxxxx otherwise. The difference between the schemes using rehash bits and LRU bits can be shown in handling a sequence of references as in Table 2.2: a_000010, b_100010, c_000010, and b_100010, assuming that a, b, and c are the tags and the 6-bit binary numbers represent the primary set IDs. After the second request in the rehash-bit scheme, both rehash bits are off indicating both locations are now a primary location. As a result, the third request c_000010 will replace b_100010 instead of the LRU line a_000010 and cause a miss to the fourth request. Moreover, an additional search and one extra line movement are

Table 2.2. Comparing column-associative caches using rehash bit and LRU bit

Scheme	Location	Initial Value	Contents after References			
			a_000010	b_100010	c_000010	b_100010
LRU bit	Cache set: (000010)	x	x ← a	a	a ← c	c
	(100010)	y	y ← x	x ← b	b	b
	LRU bit	0	0	1	0	1
	Search 2nd set	–	yes	no	**no**	no
	Replaced line	–	y	x	**a**	(hit)
Rehash bit	Cache set: (000010)	x	x ← a	a	a ← c	c
	(100010)	y	y ← x	x ← b	b ← a	a ← b
	Rehash bit: (000010)	0	0	0	0	0
	(100010)	0	1	0	1	0
	Search 2nd set	–	yes	no	**yes**	no
	Replaced line	–	y	x	**b**	a

also observed for the rehash-bit scheme in response to the third request. Figure 2.6 illustrates operation flow of LRU-based column-associative cache, and the numbers in the figure indicate latency.

The advantages of the LRU-based search algorithm for 2-way column-associative caches are intuitive. When the column associativity is greater than two, the average number of searches to find the cache line becomes an important performance issue because hit time of the secondary location increases as the number of searches increases. However, there are two conjectures that will overcome the adverse performance impact of the LRU-based column-associative search. First, a very high percentage of the requested lines would be found in the primary locations. This is true for most of the SPEC95 benchmarks in which more than a 96% of the total cache hits is directly to the primary locations (details in Section 2.6). Second, according

to the LRU sequence, the average number of searches would be small because of the program locality.

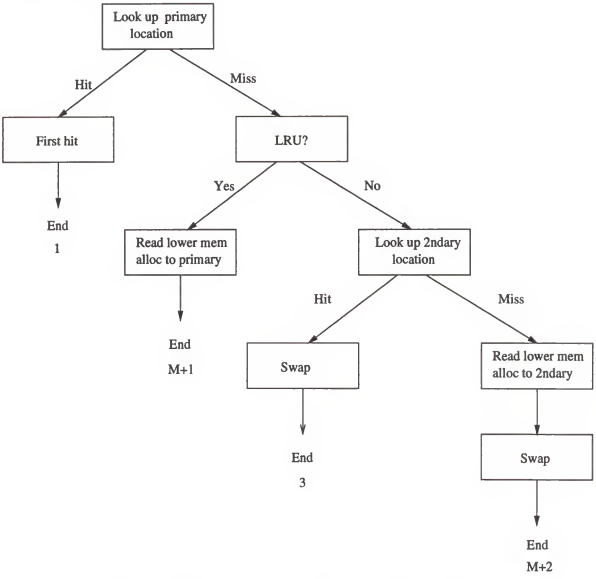


Figure 2.6. LRU-based column-associative cache operation

2.5 Data Swapping

One major criticism against the multiple-access caches is due to the excessive data movement between the primary and the secondary locations. The cache lines in these two locations are swapped upon a hit to the secondary location in order

to improve the hit ratio to the primary location. When a miss occurs and the line in the primary location is more recently used, instead of being evicted, the primary line is moved to the secondary location to replace the least recently used line. This additional data movement demands higher cache bandwidth and complicates the multiple-access cache design. An indirect access mechanism has been considered to eliminate such extra data movement. In Kessler et al. [20], a MRU-bit array is maintained and accessed beforehand to determine which location should be probed first. This indirect access may lengthen the cache access time, and that is more suitable for second-level(L_2) caches [41] where the access time is less critical. Furthermore, unlike the direct-mapped hashing that targets all the cache lines in the first attempt, the MRU prediction scheme [20] limits the first target to a half of the cache lines since there is one MRU for each pair of lines in the primary and the secondary locations.

Due to the fact that the indirect access may lengthen the cache access time, it is essential to examine the critical path in the cache access carefully. A cache access typically goes through two concurrent paths. The *tag path* determines the cache hit/miss and selects a line within the set for set-associative caches. The *data path* drives the selected data out of the data array and sends the data to the execution unit. Based on the cache timing model *Cacti* [40], it can be found that the tag path is significantly longer than the data path for both the direct-mapped and the set-associative caches. This imbalanced delay provides an alternative solution to swap, i.e., swap only the tags but keeps the data array intact whenever needed. The cache access time will remain the same as long as the indirect access to the data array is

not on the critical path. This solution is also very useful when the large data SRAM is not integrated on the same chip with the tag array and other control logics.

As shown in Figure 2.7, a new *select-bit* array is included in the data path. The value of select-bit determines the location of the data which is associated with the tag of the current request. In this design, the tag and the data paths are independent unlike the original direct-mapped cache. The tag path, typically being the critical path in cache access, remains unchanged. The data path, on the other hand, requires an indirect access through the select-bit array. The two corresponding select-bits are always complemented when the tags of the primary and the secondary locations are swapped. In case of a cache miss, the corresponding select-bits may also be complemented to allow the requested cache line to be placed directly into the secondary location without first moving the line from the primary location to the secondary location to make a room for the requested line. With proper data buffering, this scheme can also eliminate the additional access to the data array when the requested data are in the secondary location. Note that a single select-bit per each direct-mapped index is sufficient to indicate whether the swapping of the data is necessary for each pair of primary and secondary locations to save space as well as to simplify the update of the select-bit.

The data array is organized as a 2-way set-associative design in which each pair of lines of primary and secondary locations is placed in the same set. The data fetched out of the data array are selected according to the value of the select-bit. The 2-way set-associative design allows overlapping the select-bit access with the data array

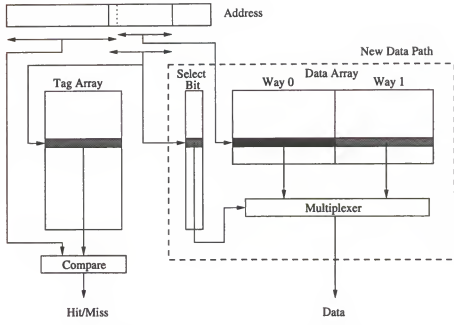


Figure 2.7. Indirect data array access

access. With proper data buffering, this scheme can also eliminate additional access to the data array when the requested data are in the secondary location.

2.6 Performance Evaluation

The performance evaluation for the LRU-base scheme and rehash functions is based on trace-driven simulation of the SPEC95 benchmark suite. Two basic metrics, miss ratio and average memory access time, are considered. The timing simulation for data swapping is experimented using *Cacti* [40] timing model.

2.6.1 Simulation Model

Simulation of separate instruction L_1 and data L_1 caches is performed. The size of the L_1 caches ranges from 8KB to 64KB. The conventional direct-mapped and other enhanced multiple-access caching schemes are applied to separate L_1 caches. These L_1 caches are backed up by a 512KB 4-way set-associative unified level 2

(L_2) cache. The line size of L_1 and L_2 caches is 32 bytes and inclusion property is enforced between L_2 cache and L_1 data cache. Further, it is assumed that a memory hierarchy system returns the critical word first and the remaining words in time for any subsequent access.

In order to compute the average memory access time, the cache miss penalties at various cache levels are considered. Without going into a detailed timing analysis, an estimation of these penalties is based upon some simplified assumptions and the general trend of current microprocessors. A hit time of 1 cycle to a conventional direct-mapped L_1 cache is assumed. If the memory request hits in L_2 cache, it takes 10 cycles to satisfy the request. When the request misses both L_1 and L_2 caches, the total access delay is 50 cycles. For various multiple-access caches, an extra delay is encountered when the requested data exist in a secondary location. A hit time of 2 cycles for the secondary location is charged since the processor pipeline is increasingly complex and difficult to turn around. Note that the search of the secondary locations is not added to L_1 cache miss penalty because L_1 cache miss can be triggered once the requested line is not present in the primary location. In other words, the delay of searching the secondary locations can be overlapped with L_1 cache miss penalty.

Shade tool [39] from Sun SPARC/Solaris environment is used in this experimentation to trace the SPECint95 and the SPECfp95 benchmarks. The standard SPEC95 input files are used. In order to avoid the initialization phase and capture the essential characteristics of these applications, the first 2 billion instructions are

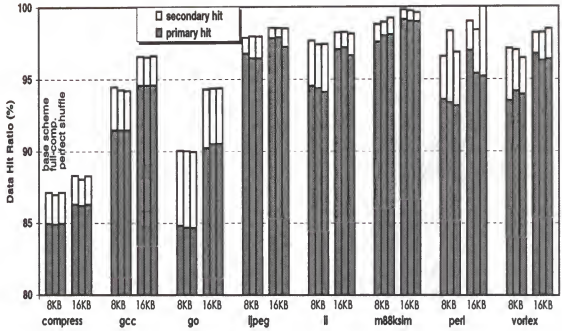


Figure 2.8. Data cache hit ratios with three different rehash functions (SPECint95)

ignored and the statistics of following 2 billion instructions are collected after the caches are warmed up.

2.6.2 Comparison of Rehash Functions

Figure 2.8 and Figure 2.9 plot the data cache hit ratios to the primary and the secondary locations for the SPECint95 and the SPECfp95 programs under three rehash functions: flipping the highest index bit (referred as the *base* scheme), complementing all index bits, and the perfect-shuffle mapping scheme. Note that the LRU replacement is used in all the simulations. The three rehash functions show comparable hit ratios for all integer programs as well as a majority of the floating-point programs. However, for benchmarks *swim*, *tomcatv*, and *wave5* from SPECfp95, the two enhanced rehash functions make drastic improvement in the hit ratios over the

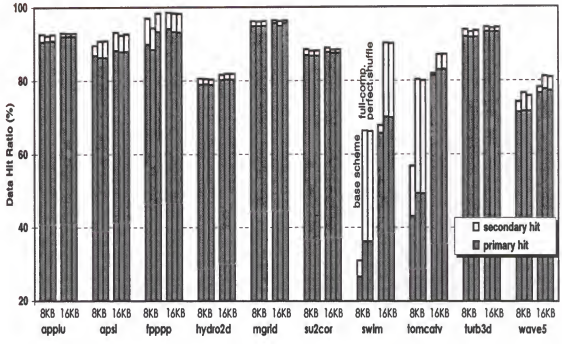


Figure 2.9. Data cache hit ratios with three different rehash functions (SPECfp95)

base scheme. For 8KB caches, for instance, the hit ratios are improved from 31.0%, 56.8%, and 74.2% to 66.4%, 80.3%, and 76.7% using the full-complement function, and to 66.3%, 80.1%, and 76.0% using the perfect-shuffle function for the respective three floating-point programs.

A deeper analysis reveals that the constant-stride memory references are very common in these programs. It can be seen that a repeated snapshot of a memory reference sequence: $..., a_1, ..., a_2, ..., a_n, ...$ throughout the program execution, where all a_i are allocated to the same set in a set-associative design. As long as n is greater than the set associativity, heavy conflict misses occur and the misses deteriorate the cache performance. The full-complement or the perfect-shuffle scheme, on the other hand, can provide a different alternative location for each a_i. Such heavy conflict

misses may disappear when the number of targets to each direct-mapped location is less than or equal to two among the nearby memory references.

Figure 2.10 and Figure 2.11 summarize the average hit ratio and the average memory access time for the SPECint95 and the SPECfp95 respectively using various rehash functions. The base scheme performs very well under the SPECint95 programs. The hit ratio and the average memory access time for the base scheme and other proposed rehash functions are very close. The bit-wise exchange schemes show a little lower performance. This is due to the identical mapping such that a number of primary locations does not have the associated secondary locations. For 16KB caches, the average memory access time for the base, the full-complement, the perfect-shuffle, the skew-shuffle, the high/low exchange, and the mirror exchange are 1.405, 1.415, 1.398, 1.399, 1.421, and 1.418 respectively.

The performance difference between the base and other rehash functions becomes more evident under the SPECfp95 programs especially when the cache sizes are small. Again, this is mainly because unusually high conflict misses for *swim*, *tomcatv*, and *wave5* under the base scheme. Among the proposed rehash functions, the full-complement and two shuffle schemes perform better than the bit-wise exchange schemes. For 8KB caches, for instance, the hit ratios are 80.1%, 85.9%, 86.1%, 86.2%, 85.6%, and 85.3% and the average memory access times are 4.880, 4,428, 4.415, 4.408, 4.453, and 4.468 for the base, the full-complement, the perfect-shuffle, the skew-shuffle, the high/low exchange, and the mirror exchange respectively. Note

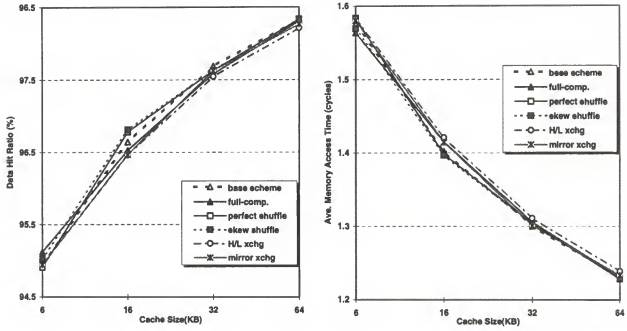


Figure 2.10. L_1 data cache performance with various rehash functions (SPECint95)

that the difference becomes less significant in larger caches because there are fewer conflict misses in larger caches.

2.6.3 Comparison of Search and Replacement Scheme

In this subsection, the LRU-based scheme is compared with both 2-way multiple-access caches and recently proposed index-vector scheme which extends 2-way set associativity.

2-Way Column-Associative Caches

Figure 2.12 and Figure 2.13 show the data cache hit ratios to the primary and the secondary locations of 2-way hash-rehash, rehash-bit, and LRU-based column-associative caches using the SPECint95 and the SPECfp95 benchmarks respectively. The corresponding average memory access times in terms of CPU cycles are shown

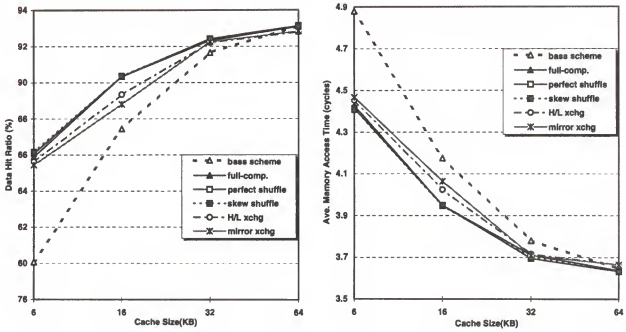


Figure 2.11. L_1 data cache performance with various rehash functions (SPECfp95)

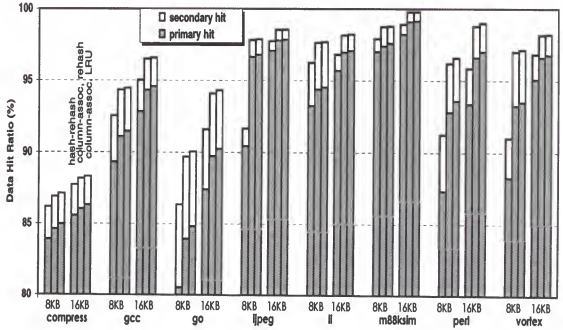


Figure 2.12. 2-way column-associative data cache hit ratios (SPECint95)

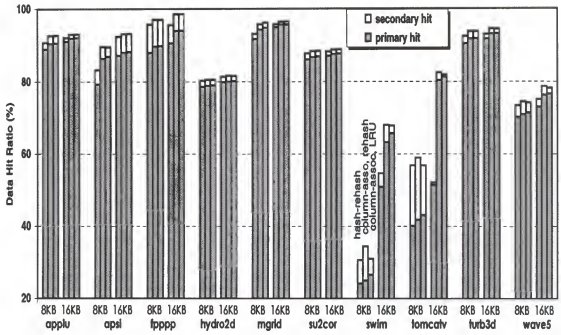


Figure 2.13. 2-way column-associative data cache hit ratios (SPECfp95)

in Figure 2.14 and Figure 2.15 for 8KB and 16KB caches. In general, the integer programs exhibit higher hit ratios than the floating-point programs. For column-associative caches, both the LRU-based and the rehash-bit schemes perform much better than the hash-rehash scheme. The LRU-based scheme, which has the same hit ratio as 2-way set-associative cache, shows a little advantage over the rehash-bit scheme. Note that a majority of hits is found in the primary locations for all three caching schemes. For example, in 16KB caches, the average percentage of the hit to the primary locations for the hash-rehash, the rehash-bit, and the LRU-based schemes are 98.0%, 98.0%, and 98.2% for the SPECint95, and those for the SPECfp95 are 97.0%, 96.9%, and 97.6% respectively.

The rehash bit makes marked improvement over the simple hash-rehash scheme for all the programs. This is especially true for the integer programs and the improvement is more significant for smaller caches. In 8KB caches, for instance, the average memory access times are improved by about 2.7%, 8.9%, 12.9%, 28.4%, 8.6%, 5.5%, 24.5%, and 26.1% for the 8 integer programs *compress, gcc, go, ijpeg, li, m88ksim, perl,* and *vortex* respectively. Note that *swim* and *tomcatv* have very low hit ratios. For 16KB caches, the rehash-bit scheme makes an abrupt improvement. This is due to the fact that these two programs show high conflict misses especially for smaller caches. The lack of precise LRU replacement of the hash-rehash scheme hurts the hit ratios badly in 16KB caches. Note also that benchmark *hydro2d* has much better hit ratio compare to either *swim* or *tomcatv*, but its average memory access time is a little worse. Moreover, the average memory access time of *hydro2d* is almost irrelevant to the L_1 cache size and caching scheme. A deeper analysis reveals that *hydro2d* has extremely high L_2 miss ratio, about 15.4%, which dominates the overall memory access time.

The simple LRU-based scheme shows marginal improvement over the rehash-bit scheme. For example, the average memory access times are improved from 2.36, 1.64, 2.05, and 1.41 to 2.33, 1.62, 2.00, and 1.36 for 8KB data caches under benchmarks *compress, gcc, go*, and *perl*. The improvements for the remaining SPECint95 are small. For some floating-point programs, such as *swim, tomcatv*, and *wave5*, however, the LRU-based scheme performs slightly worse than the rehash-bit scheme. This is again due to the high conflict misses in these programs.

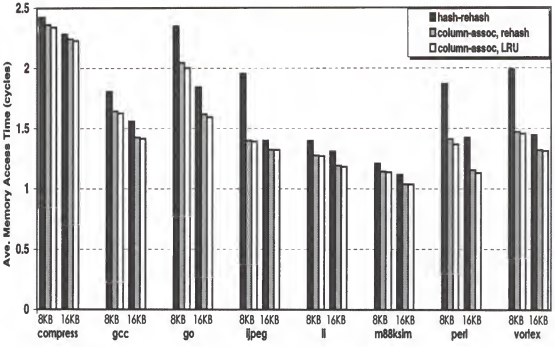


Figure 2.14. Average memory access time of data references (SPECint95)

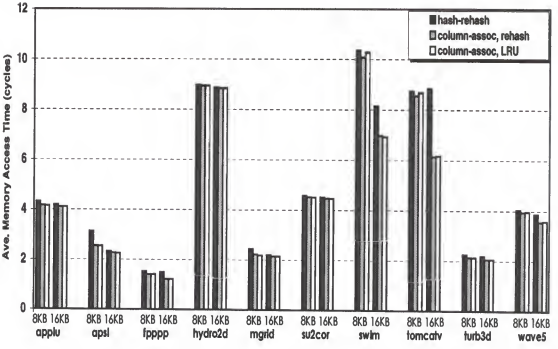


Figure 2.15. Average memory access time of data references (SPECfp95)

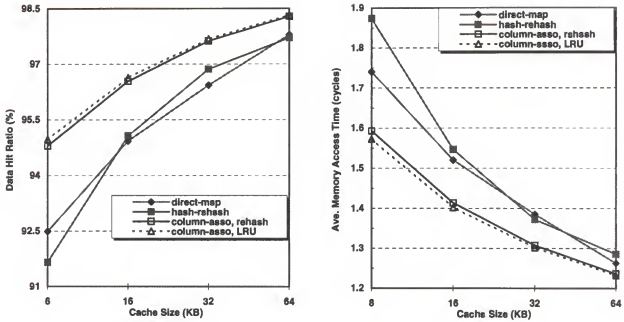


Figure 2.16. L_1 data cache performance under SPECint95

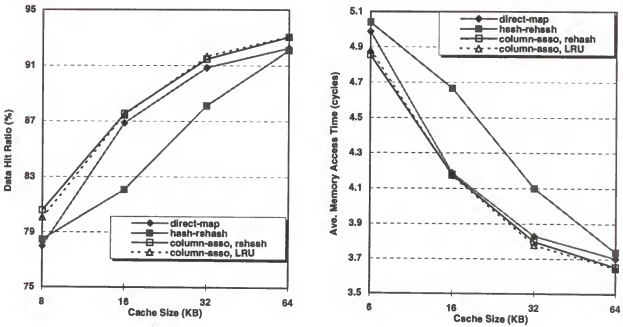


Figure 2.17. L_1 data cache performance under SPECfp95

Figure 2.16 summarizes L_1 data cache performance using the direct-mapped and three enhanced column-associative caching schemes by arithmetic means of the SPECint95 programs. As observed, the simple hash-rehash scheme does not always improve the hit ratio of the direct-mapped cache. Due to the extra delay in accessing the lines in the alternative location, the hash-rehash scheme has the worst average memory access time especially for small caches. On the other hand, both the rehash-bit and the LRU-based scheme show significant improvement over the conventional direct-mapped cache. Again, the LRU-based scheme has a little edge over the rehash-bit scheme. Taking 16KB caches as an example, the average memory access times are equal to 1.521, 1.547, 1.414, and 1.402 cycles for the direct-mapped, the hash-rehash, the rehash-bit, and the LRU-based caches respectively.

For the SPECfp95 in Figure 2.17, the hit ratios of the hash-rehash scheme are much worse than those of the direct-mapped caches of the size 16KB and 32KB. This is because *swim* and *tomcatv* have extremely poor hit ratios under the hash-rehash scheme. It is also observed that the performance improvement of the rehash-bit and the LRU-based schemes becomes less significant. The main reason is the 2-way set-associative cache can only marginally improve the hit ratios over the direct-mapped cache for a majority of the floating-point programs.

4-Way Column-Associative Caches

Comparison of the LRU-based scheme with the index-vector scheme for 4-way column-associative caches is shown here. Note that since the LRU replacement policy is used in both schemes, and the most-recently-used line is always swapped to or

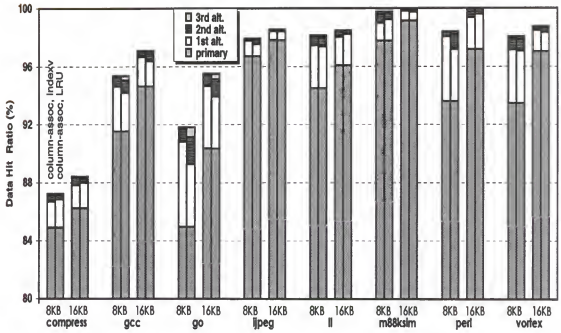


Figure 2.18. Hit ratios of 4-way column-associative data caches (SPECint95)

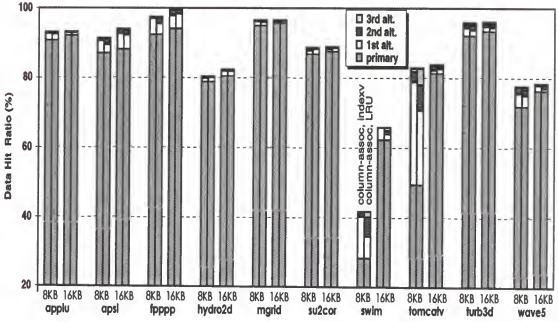


Figure 2.19. Hit ratios of 4-way column-associative data caches (SPECfp95)

placed into the primary location, the hit ratios to the primary location and the overall hit ratios of the two schemes are identical. The different hit ratios to three alternative locations separate the performance of these two schemes as shown in Figure 2.18 and Figure 2.19. For a majority of the programs, the index-vector scheme provides a shorter search path. In 8KB caches, for instance, the hit ratios to three alternative locations for benchmark *gcc* are 3.11%, 0.68%, and 0.06% under the index-vector scheme while the hit ratios are 2.69%, 0.86%, and 0.30% under the LRU search sequence. Similarly for *go*, the respective hit ratios are 5.84%, 0.95%, 0.05% and 4.31%, 1.86%, 0.67%. The difference between the two schemes is more significant for *swim* and *tomcatv* with 8KB caches. This is due to the fact that the index vector eliminates useless search. In these programs, the index-vector scheme turns out to be more effective than the LRU-based scheme which follows MRU to LRU sequence. Nevertheless, for the rest of the programs, such differences are very minor. In fact, there are a few programs, such as *compress* and *su2cor*, where the LRU-based search outperforms the index-vector scheme slightly. Also, mixed performance results between the two cache sizes are observed for benchmarks *ijpeg, li, m88ksim, perl, fpppp*, and *tomcatv*.

The average memory access times of 4-way column-associative caches for the SPECint95 and the SPECfp95 are plotted in Figure 2.20. For comparison purpose, the results of the 2-way LRU-based scheme are also included. As can be observed, the performance of the LRU-based and the index-vector scheme is almost the same for both the SPECint95 and the SPECfp95. Due to the extra delay in finding the

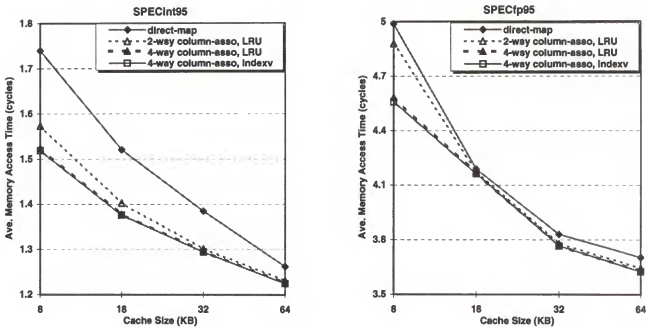


Figure 2.20. Average memory access times of column-associative caches

requested line, increasing set-associativity makes very marginal improvement except for smaller caches. The results also confirm that the column-associative caches show more performance improvement under the integer programs than under the floating-point programs.

2.6.4 Data Swapping

Table 2.3 summarizes the estimated timing of the tag and the data paths in the proposed 8KB and 16KB caches using *Cacti* [40]. In the simulation, scaling of .5um technology is assumed. Also, the following assumptions are considered: the line size is 32 bytes, and 64-bit data are fetched for each memory request. In view of the current trend, 40-bit physical address size is also assumed. In order to approximate the delay of the select-bit array, we also simulate the address with only 16 bits. In this case, the widths of the tag are 3 and 2 bits respectively for the 8KB and 16KB

Table 2.3. Timing estimation for direct-mapped and set-associative caches (unit: ns)

		direct-mapped		2-way		4-way	
size	addr	tag(comp)	data(out)	tag(comp)	data(out)	tag(comp)	data(out)
8KB	40 bits	**3.56**	3.54	5.01	**3.31**	4.98	**3.02**
	16 bits	2.55(.82)	2.69(.33)	3.46(.85)	3.30(.48)	3.52(.88)	3.02(.65)
16KB	40 bits	**3.98**	3.73	5.44	**3.93**	5.52	**3.74**
	16 bits	2.85(.78)	2.99(.35)	3.62(.83)	3.92(.54)	3.82(.85)	3.74(.65)

direct-mapped caches. Moreover, the delay of the comparator (comp) on the tag path and the delay of the output bus driver (out) on the data path are listed separately. These two delays should be excluded in identifying whether the access of the select-bit is on the critical path.

Two observations can be found from the estimated results. First, the direct-mapped tag path is a little bit longer than the 2-way and 4-way set-associative data paths as highlighted in Table 2.3. Given the fact that the set-associative data path is not on the critical path, the gap may be a little bigger by further optimizing the data path. Second, the delay of fetching the select-bit and driving the selection multiplexer is shorter than fetching the target data out of the data array. As observed, when the address is reduced to 16 bits, the delay of the direct-mapped tag path becomes much shorter than the delay of the 2-way set-associative data path. In fact, the select-bit fetched out-of the select-bit array will drive the multiplexer directly without going through a comparator as in the normal tag path. As a result, the estimated select-bit delays are only 1.73(2.55-0.82)ns and 2.07(2.85-0.78)ns for 8KB and 16KB caches. In contrast, the respective delays of fetching the data without counting the delay of driving the data to the output bus are about 2.82ns and 3.38ns. From the first-cut

timing estimation, it is shown that the proposed indirect-access of the data array by a small select-bit array will not be on the critical path in cache access.

2.7 Summary

Caches are critical components in high-performance processors and multiple-access caches attempt to achieve high hit ratio while maintaining fast access time. Three major design issues of multiple-access caches are discussed in this chapter. Those issues are rehash functions which direct the location of secondary cache access, search and replacement algorithm which strive to have lower miss ratio while minimizing the number of searches, and data swapping issue. New approaches or solutions to each of the issues are presented and simulations are performed to validate. The evaluation results reveal the followings. More randomized rehash functions with minimal hardware operations can dramatically reduce cache miss ratios in some of the scientific programs. The LRU-based multiple-access caches with new search and replacement algorithm outperform the conventional ones. Even though the improvement is not so significant, the new search and replacement algorithm merits its simplicity and cleanness. Also, comparable performance of the proposed search and replacement with conventional index-vector scheme for 4-way multiple-access caches suggests that index vector scheme is not a right direction. Finally, indirect data array access mechanism can alleviate data swapping overhead without extra cache access delay.

CHAPTER 3
EARLY-RESOLUTION OF BRANCHES AND LOADS

3.1 Introduction

Processor pipelining overlaps multiple instruction executions by issuing an instruction at every cycle. The superscalar microarchitecture with duplicated functional units and their control mechanism, on the other hand, is able to issue more than one instruction per cycle within a single instruction execution window by exploiting instruction level parallelism of the application program. However, due to control and data dependencies in the instruction stream, this conventional approach can only discover a limited parallelism unless control and data speculation are performed. Therefore, the effectiveness of this approach is constrained by the accuracy of the speculation [27, 22], and the performance of speculative processors is more sensitive to the prediction accuracy because the impact of incorrect prediction will be higher. But, even the very accurate *branch prediction* schemes [11] show that the behavior of certain classes of branches is very difficult to predict, and the accuracy of *load prediction*—either load address or load value—is very poor especially for integer-intensive programs.

In this dissertation, a different direction to achieve high prediction accuracies for both branch and load instructions is presented. The rationale is to execute the

branch/load instructions early by data-flow link between dependent instructions, and produce the expected results for prediction values ahead of normal instruction flow. This new approach dynamically keeps track of producer to consumer relationship by efficiently building a data-flow graph for prediction.

The performance evaluation using a simple machine model developed on the top of the *Simplescalar* simulation tool [6] shows very promising results. For the selected SPECint95 programs, the prediction accuracy by early resolution of branch instructions is significantly better than traditional branch prediction and that of load instructions exhibits accuracy far beyond the existing load address/value prediction schemes.

3.2 Inherent Limitations of Control and Data Speculation

Processor pipelining overlaps multiple instruction executions by exploiting instruction level parallelism to improve CPI or cycle time of computer systems. Figure 3.1 shows an example of a single-issue 5-stage pipeline and Table 3.1 shows a sequence of instruction flow in the pipeline. At *IF (Instruction Fetch)* stage, an instruction is fetched and next PC (Program Counter) is determined. *ID (Instruction Decode)* stage decodes the fetched instruction and reads register values from register file. *EX (Execution)* stage executes the instruction depending on the type of the instruction. *ME (Memory)* stage accesses data cache if the instruction needs data reference. Finally, the result of the instruction is written back to register file at *WB (Write Back)* stage.

As can be seen from Table 3.1, *data hazard* prevents the next instruction from being executed since the next instruction depends on the result of the previous instruction, and *control hazard* prohibits the sequentially-fetched next instruction from being executed since the previous instruction's result can redirect the instruction sequence. In this figure, specifically, data hazard and control hazard are referred to as *load hazard* and *branch hazard* respectively. Both are classical pipeline hazards that may incur severe penalty in multiple-issue superscalar designs in spite of the advanced dynamic instruction scheduling and out-of-order execution with high-performance cache memory to hide the latency. A study by Lipasti and Shen [23] shows, as shown in Figure 3.2, the diminishing performance returns are occurred in superscalar processors due to the pipeline dependencies.

Significant progress has been made recently to alleviate the adverse performance impact due to both control and data dependency. *Speculation*, which allows the execution of an instruction with the predicted outcome of the previous instruction, has been explored to exploit more parallelism. To be more effective by speculation, however, the accuracy of prediction for the instruction outcome plays a a crucial role because incorrect speculation involves overhead to recover the machine state.

Various accurate *branch* path prediction schemes, reaching above 90%, have been proposed in the studies [42, 26, 25], and they follow the observation that the outcome of a branch is highly correlated with the previous outcomes of the same branch as well as the recent outcomes of nearby branches. However, recent studies also show that the behavior of certain classes of branches is very difficult to predict [11] suggesting the

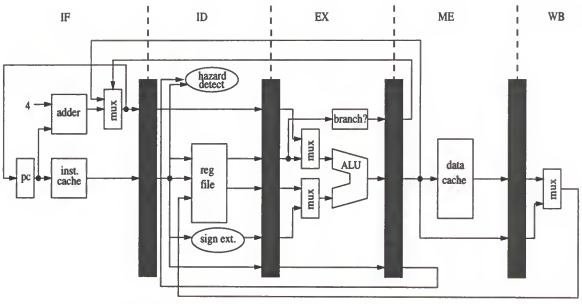


Figure 3.1. A simple 5-stage pipeline example [16]

above mentioned correlation-based predictor is closely reached to its upper bound. For example, elaborate correlation-based predictors such as the 2-level per-address predictor [43] and hybrid predictor [25] do not work better than the ideal static predictor on 55% of the branches in the integer-intensive programs of the SPEC95 benchmark suite [37]. Furthermore, about 17% of these branches is not heavily biased towards either the true or false paths and suffer from high misprediction rate [43].

Load instructions, on the other hand, exhibit inherent source of latency because the loads need *effective address* calculation before memory access can start. Therefore, the load result is available at the later stage of pipeline forcing the load-dependent instruction be stalled in the pipeline. For example, in Table 3.1, load-dependent instruction *sub* at clock 4 needs a stall because the load outcome is not

Table 3.1. A MIPS [16] instruction sequence and its execution on the pipeline

```
        lw  $r1,4($r2)          # $r1 <-- memory[4+$r2]
        sub $r3,$r1,$r4         # $r3 <-- $r1-$r4
        beq $r3,$r0,Target      # if ($r3==$r0) goto Target
        sw  $r3, 4($r2)         # memory[4+$r2] <-- $r3
             ...
Target: add $r5,$r6,$r7         # $r5 <-- $r6+$r7
        or  $r8,$r9,$r10        # $r8 <-- oring of $r9 and $r10
```

cycle	IF	ID	EX	ME	WB	remarks
1	lw					
2	sub	lw				
3	beq	sub	lw			load and control dependency
4	beq	sub	–	lw		sub is stalled(lw forwards $r1 to sub)
5	–	beq	sub	–	lw	IF is stall(sub forwards $r3 to beq)
6	–	–	beq	sub	–	determine branch decision
7	add	–	–	beq	sub	
8	or	add	–	–	beq	

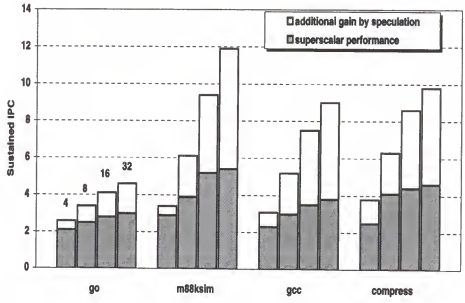


Figure 3.2. Sustained IPC of superscalar processors(4 to 32 issues) [23].

ready at this time. The effect of load hazard for the multiple-issue superscalar processors is even more significant. Simple hardware scheme such as forwarding does not solve the load hazard.

In an early work, Hua et al.[18] proposed mechanisms to overlap address generation with cache access, i.e., prior to the address generation cycle, the base register contents are used to predict the real index bits needed to access the cache. Austin et al.[3] suggest another mechanism to predict the cache index bits by simply *oring* the index portion of the base address and the offset because the oring can approximate bit-wide addition of two sources of base and offset. Instead of waiting the result of the effective address calculation, a newly proposed SAM cache [24] determines cache hits and misses by the individual base and offset directly to the decoder of the cache RAM array.

One alternative approach is *load address prediction* as suggested in the literatures [10, 14, 4]. This approach comes from the conjecture that the load address is likely to remain the same across the multiple instances of the same load when the data dependency is maintained through memory. In addition, when the load instruction is used to access regular data structure such as arrays, the load address tends to change with a constant stride. The accuracy of load address prediction is relatively high especially for the floating-point intensive programs because of their uniform memory access pattern. However, a recent study by Black et al. [5] shows that the accuracy for the integer-intensive programs is still very poor, only about 51% on the average, even with a highly sophisticated predictor. These results suggest that a significant

portion of the loads in the integer-intensive programs encounters irregular address changes and difficult to predict.

Recently, based on the observation that load values exhibit some locality, Lipasti et al. [21] and Lipasti and Shen [22] propose predicting the load value of an instruction using the last load value by the load instruction. This *value prediction* can be performed as soon as the load instruction is fetched so that the dependent instructions can be executed speculatively without any stall. Unfortunately, the studies also reveal that the accuracy of load value prediction is quite low. Load value prediction is inherently inaccurate due to lack of correlation between successive load values by the same instruction. For example, if the address of a specific load remains the same across multiple executions, we would expect the corresponding memory contents have been modified. If the address is changed whenever the load is executed, the possibility of obtaining the same value should be very slim. There are certain specific instances such as sparse matrix computation, register spill code, memory alias resolution, etc., where the loads do exhibit value locality [21]. However, to a large extent, value locality tends not to be prevalent in general programs.

As described above, the IPC is limited to a very small number even in very wide-issue superscalar processors due to the pipeline dependencies as shown in Figure 3.2. In order to overcome this limitation, speculative execution is widely accepted in modern microprocessor design. However, the accuracy of speculation for branches is very close to its upper bound in current elaborate branch predictors and that of loads is very poor because of program behavior. Thus, in spite of many efforts to

reduce latency by branch and load instructions, the performance penalty can still be severe especially in multiple-issue processors. In this chapter, a different direction to improve IPC is presented by achieving high prediction accuracies for both branch and load instructions which are suppose to be executed speculatively. The rationale is to execute the branch/load instructions early by a data-flow graph between producer and consumer which has been established over the program execution, and produce the expected results ahead of normal instruction flow.

3.3 Architectures to Improve IPC

There have been several advanced microarchitecture proposals to improve the IPC for future microprocessors. A brief survey of the proposals is explained in this section.

The advanced superscalar processors [27], for handling wide-issue of 16 to 32 instructions, use trace cache to utilize a logically contiguous instruction sequences and multi-hybrid branch predictor to aid context switching and indirect jumps. The super-speculative processor [23] enhances the wide-issue superscalar processor with aggressive data speculation to break the data dependence chain and to alleviate the load stalls. The idea of the super-speculative processor comes from the observation that producer instructions in the dependency chains create highly predictable values allowing consumer instructions to speculate with the predicted values. The trace or multi-scalar processor [36, 35] breaks the program into traces or tasks. Multiple hardware cores execute multiple traces in parallel to achieve high IPCs. Register data-flow is maintained by sending register results from the producer tasks to the

consumer tasks, while memory data-flow is speculative until the store and the load addresses are available from the producer and consumer tasks. The simultaneous multi-threaded processor [9] shares an aggressive pipeline among multiple tasks to utilize the pipeline efficiently. It actually combines the idea of wide-issue superscalar and multi-threaded processors, and attempts to issue multiple instructions from multiple threads in a single cycle. Chip multiprocessors [15] implement multiprocessors on a single chip and run parallel programs or multiple independent tasks on these processors. Chip multiprocessors can be underutilized if a program does not exhibit sufficient parallelism. Finally, the vector IRAM processor [28] couples vector processor with high-bandwidth and low latency DRAM on the same chip to achieve high performance.

The data-flow computation model has been around for more than 20 years [8]. The basic data-flow concept has been applied in a recent work for prefetching the link-based data structures [30]. The link-based data access, also known as the pointer chasing problem, creates a sequence of loads which has tight register data-flow dependencies between each other. The proposed method builds dynamic data dependence list among the link-based data structures. The dependence-based prefetch can be triggered based on the existing list. To overcome the limitation of correlation-based branch prediction schemes, Farcy el al. [12] used a similar method for branch resolution.

The decoupled access/execute architecture [34] separates memory access unit from the normal execution unit. Two separate instruction streams are generated by

the compiler, one for memory access and the other for data manipulation. Hardware send/receive queues are used to satisfy the memory data dependencies between two instruction streams. In this design, it is conceivable that the memory access stream can run ahead of the execution stream to hide the load latency without encountering the constraint imposed by a single execution window.

3.4 The Proposed Microarchitecture

There are two sources of data dependencies during the execution of a program. The first is through registers and it is commonly referred to as *register data-flow*, the second is through memory and also known as *memory data-flow*. With regard to the register data-flow, the branch outcome and the load address are likely to be changed when the corresponding source operand registers or the base/index registers are updated. Therefore, a straightforward approach for reducing the stalls caused by branches and loads is to re-evaluate the branch condition and re-calculate the load address upon an update of their source registers. The memory data-flow, on the other hand, can only affect the load results and can be detected by matching the current load address with the outstanding store addresses. In principal, a load may be triggered upon the completion of the nearest store that addresses the same memory location but the load can generally skip memory read operation by receiving the results forwarded by a store instruction.

Early execution of an instruction can be initiated beyond the normal execution window if the dynamic register and memory data-flow links have been established from the instruction that updates a register or memory location to the branch or the

load instruction that uses the new value. This is similar to the data-flow principle in that an instruction is triggered whenever its operands are ready [8]. As a result, the branch outcome and the load value may be available before the respective instructions are actually ready to be issued according to program order.

A newly proposed microarchitecture in this dissertation employs three data structures to establish the register data-flow links and to trigger the branch/load instructions early. Such links are built dynamically based on the recent execution paths. The *Register Update Table (RUT)* remembers the instruction address where the most recent update to each register took place. So, the size of RUT is the same as the register file. The *Data-flow Link Table (DLT)* records the data-flow links that connect more recently executed instructions, which participate register update, to a branch or load instruction where the updated register is used as one of the source registers. Finally, the *Early-triggered Reservation Buffer (ERB)* serves as the reservation station for both renaming the destination registers and initiating the execution of the early-triggered instructions when the source operands are ready. It also acts as a reorder buffer to save the results temporarily for the early-triggered instructions and to handle the recovery when a mis-triggered instruction is identified.

The algorithm for building the dynamic register data-flow links works as follows. When an instruction that involves an update of a register is decoded, the current instruction address is recorded into the RUT to replace the previous instruction address where the same register was updated. When a branch or a load instruction is decoded, the register data-flow link needs to be built and recorded in the DLT if such

a link does not already exist. The source of the link can be fetched from the RUT using the source operand register ID of the branch/load instruction. The destination of the link is simply the location of the respective branch/load in the instruction cache (I-cache). Note that there can be either one or two links depending on the number of operands in a branch/load instruction.

Early resolution of branch/load involves several steps. First, the data-flow links must be built as described earlier. Second, when a register-update instruction is fetched, the DLT is looked up simultaneously. The links found in the DLT are used not only to fetch the dependent branch/load instruction from the I-cache but also to trigger the instruction. In case the ERB is full, the early-triggered instruction at the Least-Recently-Used (LRU) position is replaced. With this mechanism, the branch/load instructions can be executed whenever the required operands are ready. Third, upon the completion of an early-triggered branch/load instruction, the early execution result is simply saved in the respective entry in the ERB and marked as ready. The respective entry of the ERB renames the destination register effectively. If a branch/load instruction has been issued by the normal issuing logic, the result is also moved to a proper entry in the regular reorder buffer to allow the branch/load instruction to be committed in program order.

Finally, when a branch/load instruction is fetched according to the program order, the ERB is accessed simultaneously. The branch/load instruction is decoded and issued normally if it misses the ERB. If the instruction is found in the ERB and the result is marked as ready, the branch outcome or load value can be obtained directly

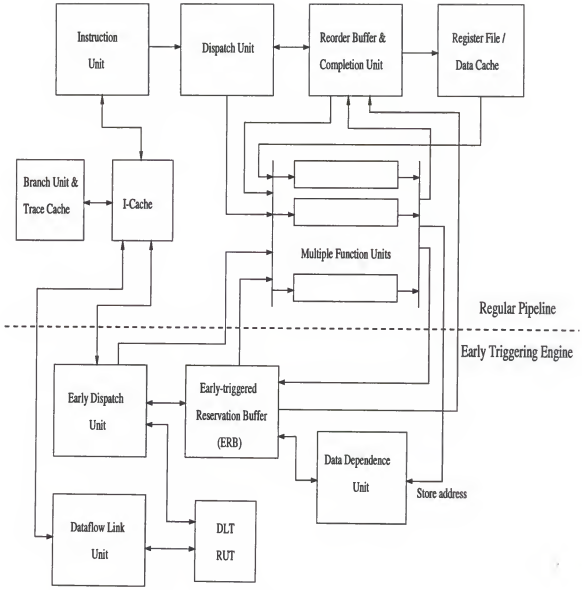


Figure 3.3. Block diagram of proposed microarchitecture

as if it is coming from the branch prediction logic or the load value prediction table. As a result, the branch outcome can be predicted more precisely and the instructions that depend on the load value can be issued without any stall. If the early-triggered branch instruction has not reached the completion stage, the normal branch prediction outcome is used to fetch the subsequent instructions and the prediction can be verified by the outcome of the early-triggered branch evaluation. For a load, an entry in the reorder buffer is allocated to accept the data from the early-triggered load. The early-triggered instruction is dropped from the ERB after it is executed normally according to program order. Note that although the branch/load instruction may be triggered beyond the normal execution window, all the instructions are committed in program order to preserve precise interrupts and to simplify recovery handling.

A block diagram of the proposed microarchitecture is presented in Figure 3.3. The newly proposed processor relies on a regular superscalar pipeline to handle the actual execution of the instructions. To use the available instruction slots more effectively, the architecture includes an additional Early-Triggering Engine (ET-Engine) that enables instructions to be executed as soon as the operands are available. There are three major functional units in the ET-Engine. The Early Dispatch Unit triggers an execution of the branch/load early from the I-cache indexed by the register data-flow links in the DLT. An entry is allocated in the ERB for the early-triggered instruction. The Early Dispatch Unit also schedules the instruction to the functional units once the required operands are ready. In addition, when the early-triggered instruction is fetched in program order, the Early Dispatch Unit searches the ERB

and merges the result into the reorder buffer. The ERB entry is freed afterwards. The Data-flow Link Unit searches the DLT and the RUT to establish the register data-flow links for the branch/load instructions. Finally, the Data Dependence Unit compares the early-triggered load addresses with the store address available in the order of program execution. When a match is found, the data obtained from the early-triggered load are replaced with the store data.

3.4.1 A Detailed Example

Figure 3.4 illustrates how the RUT, the DLT, and the ERB operate for early resolution of branch/load with a simple program segment. A simple loop example in the figure is extracted from some of the source codes of *gcc* in the SPECint95 benchmark suite. The RUT maintains the instruction addresses where the most recent updates of the respective registers have taken place. For example, register *$3* is first updated by the second instruction *addu (2)*, then updated again by the fifth instruction *addu (5)*. The latter update is reflected in the RUT by replacing the previous instruction address *addu (2)* with the new instruction address *addu (5)* in the RUT entry of register *$3*.

Both the DLT and the ERB are searched at the instruction fetch cycle. The size and set-associativity of the two tables depend largely on the performance evaluation results. When *lbu (4)* is first executed, there is no previous link pointing to this instruction. As a result, based on the last update of the source register *$3*, a link *(2)* $\rightarrow$ *(4)* is entered into the DLT(the same steps for a link *(2)* $\rightarrow$ *(4)* are occurred earlier). However, this link may not have any effect since the instruction *addu (2)*

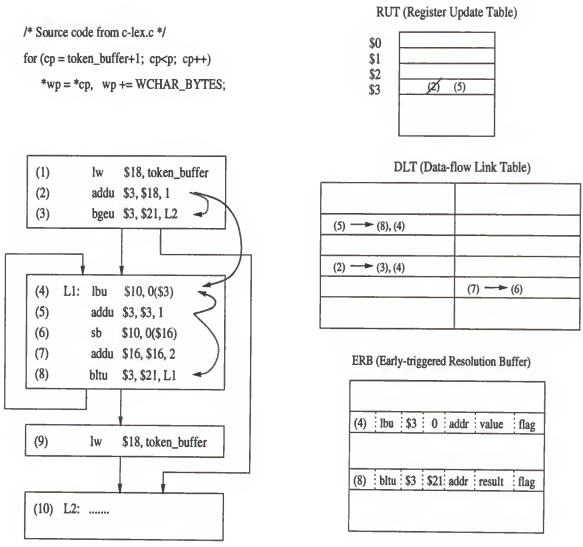


Figure 3.4. Example of early resolution of branches and loads

does not have a chance to be executed again to trigger *lbu (4)*. When *addu (5)* is first executed, the RUT is updated and this allows the new link *(5)* → *(4)* to be built when *lbu (4)* is executed again. Such a link will trigger *lbu (4)* earlier when *addu (5)* is encountered again. Similarly, the link *(5)* → *(8)* is built to allow early-triggering of *bltu (8)* by *addu (5)*.

The early-triggered *lbu (4)* allocates an entry in the ERB. Upon the completion of *addu (5)*, *lbu (4)* moves forward to the execution stage. The scheduling of *bltu (8)*

is different because it has two source registers. Although *bltu (8)* can be triggered by *addu (5)* in this example, it will never have a chance to be executed early because it must wait for the second operand. This situation can be avoided by recording the status of required operands for the early-triggered branch/load instructions. When *bltu (8)* is fetched normally in program order, a search through the ERB can confirm that the instruction has been triggered. In this case, a special condition is recorded for *bltu (8)* if the *flag* in the ERB indicates that one of the operand has never been encountered. When *bltu (8)* is triggered again in the future, it can be executed once register *$3* is available.

Early-triggered load may violate memory data-flow. For instance, it is conceivable that the store address at *sb (6)* may match the load address of the early-triggered *lbu (4)*. In this case, the load value may be incorrect due to the data dependency. A straightforward solution is to search the ERB when the store address is available from the normal pipeline. The store data are forwarded to the entry with the matched load to replace the incorrect load data. It is also possible to verify potential data dependency violation early by triggering the address computation of *sb (6)*. The instruction *sb (6)* can be triggered by the register data-flow link from *addu (7)* as shown in the DLT of Figure 3.4.

3.4.2 Implementation Issues

There are several important implementation issues that need to be studied carefully. These issues are explained as follows.

Accuracy of Triggering: The data-flow link is built dynamically based on the current execution path, which may or may not be repeated. As a result, a branch/load can be mis-triggered. There are three types of inaccurate triggers.

First, an early-triggered branch/load may never be encountered in the normal execution path. This situation may happen during the execution of a program loop. For instance, in the example of Figure 3.4, the early-triggered *lbu (4)* beyond the last iteration causes an unnecessary load. The same situation may also be encountered in procedure calls and returns.

Second, due to the dynamic execution paths, the instruction that triggers a branch/load based on the existing data-flow link may not be the instruction that makes the last update to the respective source register. One simple way to verify the correctness is to compare the results of the early execution with the results of the normal execution. However, this requires that the early-triggered instruction be executed twice. A better solution is to save the source register contents in the ERB when an instruction is early-triggered. When a branch/load instruction is ready to be issued in program order, the register contents in the ERB are compared with the current register contents. The early execution is nullified if a mismatch is found.

Third, there are cases where a branch/load cannot be triggered because the source registers are not modified between different execution instances, implying that the branch outcome or load address remains the same. Such branch/load is clearly good candidates for branch prediction or address/value prediction. With the dynamic register data-flow links, it is possible to identify this type of load/branch

more accurately. When a load is fetched that does not exist in the ERB, a link is established in the DLT. If the same link already exists, this load becomes a candidate for address or value prediction. For branches, the normal branch prediction logic can be used to predict the outcome for the branches that are absent from the ERB. We will show the accuracy of early-triggering of branch/load since it plays a crucial role in the proposed new microarchitecture.

Timing of Early-Triggering: The effectiveness of early-triggering of branch/load depends on the amount of time available for resolving them. In general, early-triggering is most effective if all the branches/loads can be resolved accurately right before they are fetched and issued in program order. However, this may not always be the case in realistic program code. For example, given the fact that the number of architecturally-defined registers is very limited, the compiler usually tries to minimize the distance between the source and the destination of the register data-flow link to free the registers sooner so as to reduce the spilling of registers. Even worse, as described earlier, it is likely that the load instruction, whose role is to restore the register contents, may not have a chance to be triggered early since the base register contents remain unchanged.

When the results of an instruction cannot be produced in time by early-triggering, one possible hardware solution is allowing the evaluation of its source operands to be triggered early. For instance, in the example illustrated in Figure 3.4, if the early-triggered *lbu (4)* does not produce the load data in time, the source of the data-flow link, *addu (5)*, can be allowed to trigger itself by establishing a new data-flow link

from itself. The completion of *addu (5)* can further trigger *bltu (8)* and *lbu (4)*. As a result, the early-triggering can move further ahead with respect to the normal execution window. However, this aggressive demand-driven data-flow execution can produce multiple instances of the same instruction in the ERB. Proper tagging of those instructions is necessary to maintain the data dependence correctly. In addition, limitations have to be imposed on this aggressive data-flow execution to keep the hardware complexity manageable.

Amount of Extra Traffic: The best scenario of the architecture is to imagine the precise early-triggering for the instruction that is soon to be executed by normal program order. But, the program behaviors, such as multiple path executions and inter-procedure calls, potentially incur extra or useless executions of loads/branches. For example, the multiple links which were built by *case statements* in C programming language as consumers will cause redundant early-triggering. A careful and clever treatment for the multiple paths need to be considered. The procedure call related issue is discussed in section 3.4.3.

Hardware Resources: The effectiveness of the proposed microarchitecture also depends heavily on the available hardware resources. For instance, additional data cache ports may be needed to accommodate the early-triggered loads. In addition, this newly proposed processor may require additional functional units, especially when the early-triggering of operand evaluation for the branch/load instructions is permitted. Chip area is also needed to implement the RUT, the DLT, and the ERB. The performance impact of the size and topology of the DLT and the ERB

needs careful evaluation. In general, a bigger DLT can maintain more register data-flow links and, therefore, trigger more branches/loads. However, it may keep the out-of-date links longer and increase the chances of mis-triggering. A larger ERB should produce better performance by reducing the likelihood that an early-triggered execution result will be replaced before it is actually used.

3.4.3 Optimizations for Register Restore

In a typical procedure call convention, a subset of the registers have to be saved and restored by the caller and callee routines. At procedure invocation, a stack frame is created not only for saving the required registers and the local variables but also for passing parameters. At procedure return, all the saved registers are restored and the stack frame is squashed. This save and restore of registers create artificial data dependencies that can greatly affect the accuracy and effectiveness of early triggering of branch/load instructions in the proposed processor.

Figure 3.5 illustrates an example of procedure call from a benchmark *go*. Both procedures *check_threatened()* and *fixurgdefarmies()* call the same procedure *killist()* during execution. As a result, links are established in the DLT to reflect the flow of data from the last update of the stack pointer *$sp*. The last update of *$sp* in *killist()* frees the stack frame, and a sequence of *lw* instructions in both *check_threatened()* and *fixurgdefarmies()* restores the respective register contents. These are indicated by the dashed arrows in Figure 3.5. In reality, only the restorations of the calling procedure should be triggered early. Therefore, the artificial data-flow links seriously reduce the accuracy of the early-triggering mechanism. This situation is exacerbated further

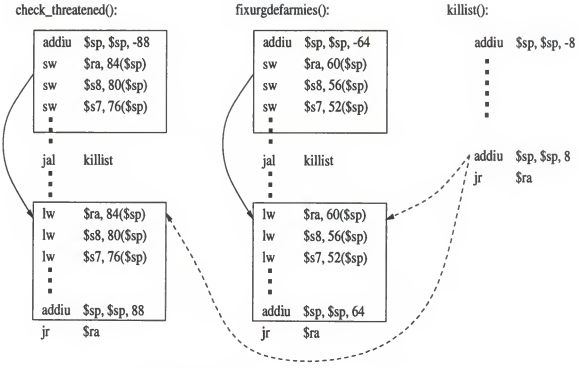


Figure 3.5. Procedure call example from *go*

when several procedures alternately invoke *killist()* because the links for a calling procedure may be replaced prematurely by more recent caller links. Such artificial data-flow links also exist in other registers. In fact, it applies to any spill code that saves and restores registers. Because of the frequency of procedure invocation, some specific optimizations will be required, which can be applied to the restoration of registers in these situations.

In this proposed processor, the destination registers are renamed using the ERB. Therefore, as indicated by the solid arrows in Figure 3.5, the restoration of registers can be triggered immediately after the registers are saved on the stack frame. In order to establish the correct link, the address of the instruction that updates the *$sp* is pushed onto a separate stack, called *link stack*, when a procedure call is encountered.

Upon returning from the procedure, the instruction address at the top of the link stack is popped and used to update the corresponding entry for the $\$sp$ in the RUT.

If the same procedure is called more than once in a nested fashion, the same set of loads for restoring the registers may have been early-triggered multiple times to restore the registers from different stack frames. In order to accommodate multiple triggerings of the same instruction in the ERB, each procedure invocation is *colored*. The color or stack depth can be maintained as a simple up-down counter that is incremented on procedure calls and decremented on procedure returns. In addition to the instruction address, the color needs to be matched when a search is initiated for the early-triggered instructions in the ERB.

While such an early restoration of the registers into the ERB eliminates the load delays associated with register restoration, it requires that the ERB be sufficiently large to hold the different register values especially for deep recursive calls. One technique for alleviating the ERB space requirement is to *defer* the early triggering of restorations when the stack depth exceeds a threshold. When this happens, the restorations at a given stack level are not triggered before the next level is invoked, instead, the restorations are initiated when the next level returns. If the link stack contains the value of the stack pointer in addition to the address of the instruction, the restorations can be triggered several levels ahead by using the stack pointer information of top few entries in the link stack.

3.5 Strength of The Proposed Architecture

The proposed microarchitecture establishes a new microarchitecture paradigm by providing an efficient way to enlarge the instruction execution window effectively with minimal additional requirements on both original instruction fetch unit and issue unit. Based on data-flow ideas, the new processor is able to trigger critical branch/load instructions beyond the current instruction window. If the early-triggered branch/load instruction is still on the critical execution path, the instruction that initiates the trigger can also be triggered early. However, in contrast to the pure data-flow machines, the demand-driven execution model in the proposed architecture is selectively applied to those instructions that are on the critical execution path in order to bound the hardware complexity. The non-critical instructions are executed normally in the regular instruction window. Although, the early resolution of branch/load is emphasized in this dissertation, the new microarchitecture can be easily generalized to allow other instructions to be triggered whenever its operands are ready.

The proposed processor is able to accomplish the demand-driven execution without any special support from the programming language or compiler. The register data-flow links are established dynamically by the hardware. The memory data-flow dependencies are checked at run time to allow early-triggering of loads. Although the new processor does not provide multiple virtual register sets as is common in the multi-threaded approaches, it performs register renaming aggressively on the early-triggered instructions to eliminate any name dependencies effectively on the critical

execution path. In addition, by restoring the register contents into the ERB early, the new processor minimizes the adverse impact of saving and restoring registers efficiently for procedure invocation.

The dynamic data-flow links and the ERB provide an additional benefit by identifying the loads with a constant address across different execution instances. These are the loads that are suppose to be predicted precisely by address/value prediction. An optimistic approach is to invoke address/value prediction whenever the load is not found in the ERB. A more conservative way is to resort to address/value prediction only when a link is present in the DLT but the instruction is absent from the ERB. By using address/value prediction only for those loads that are likely to benefit from address/value predictor, a majority of correct predictions may be captured with smaller prediction tables. In addition, the number of mis-predictions can be reduced drastically.

3.6 Performance Evaluation

Given the fact that the accuracy of early-triggering of branch/load plays a vital role in determining how viable the newly proposed architecture is, the primary focus is to find out the the correct triggers and the useless branches/loads produced by early-triggerings. In addition, the comparison of the correct early loads is shown under the new microarchitecture with the stride-based last-address and last-value prediction results. Furthermore, the branch prediction accuracy with/without the proposed early-resolution mechanism is also provided.

3.6.1 Simulation Model

The simulation model in this section is built upon the *Simplescalar* tool set [6] version 2.0. The Simplescalar architecture is a close derivative of the MIPS-IV architecture [19]. A fast functional simulator was used for the purpose this simulation. The new microarchitecture model is built around the Simplescalar instruction interpreter. After the simulator interprets each instruction, the necessary steps such as updating the RUT, building the data-flow links to the DLT, and early-triggerings of branch/load into the ERB are performed. In case that the instruction is a branch/load, the search through the ERB determines the hit/miss, and also determines whether the normal branch or the load address/value prediction is needed.

The Simplescalar tool set comes with precompiled-compiled binaries of the SPEC95 benchmark suite. Seven integer programs, *compress, gcc, go, li, m88ksim, perl*, and *vortex*, are selected to drive our proposed model. For each application, a warm up of first 200 million instructions is made for all the buffers and tables. After a warm up, the statistics based on the simulation of another 200 million instructions are collected. In this simulation, various sizes and the set-associativities of the DLT and the ERB are investigated. Both the DLT and the ERB are varied from 64 to 512 entries with the set-associativities from 4 to 16. In *large* configuration, both the DLT and the ERB are 16-way set-associative with 512 entries, while in *small* configuration, the two tables are still organized as a 16-way set-associative with 128 entries.

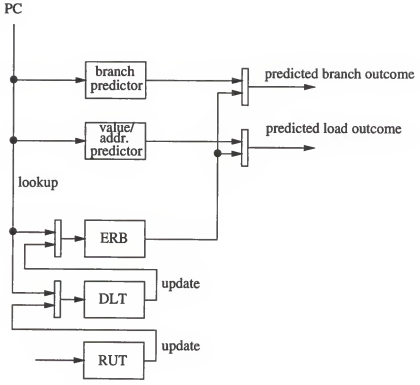


Figure 3.6. Functional block diagram of simulation model

In the simulation model, a 64-entry link stack is constructed to handle the nested and recursive calls. In addition, the gshare/bimodal hybrid branch prediction scheme with 2K-entry for each branch history table is constructed. The hybrid branch predictor is used only when a branch instruction misses the ERB. Moreover, the load address and the load value prediction tables are implemented to capture the loads which miss the ERB. For comparison purpose, stand-alone stride-based last address/value prediction mechanisms are considered. Both prediction tables contain 2K entries with 16-way set-associativity. The functional blocks of the new microarchitecture model in this simulation are illustrated in Figure 3.6.

3.6.2 Evaluation Result and Discussion

Figure 3.7 and Figure 3.8 summarize the accuracy of the early-triggered loads. The shaded regions represent the percentages of loads that are correctly triggered. The white regions on the top of the shaded regions, on the other hand, indicate the correct addresses or values predicted additionally by the respective prediction tables. The accuracy of stand-alone stride-based address/value prediction is also presented for comparison purpose. Figure 3.9 shows relative accuracy of stand-alone address predictor and address predictor with the ERB which performs prediction when a miss occurs in the ERB. Two important observations can be found from the figure. First, the early resolution mechanism produces significantly higher correct load addresses/values than the stand-alone prediction methods. For instance, the percentages of correct load values for *compress, gcc, go, li, m88ksim, perl,* and *vortex* are about 83%, 75%, 69%, 93%, 86%, 86%, and 79% respectively for small DLT/ERB configuration. For large configuration with 4 times bigger DLT/ERB, the respective accuracies become 83%, 80%, 75%, 95%, 95%, 86%, and 84%. Without the early-triggering mechanism, the value prediction accuracies based on the last value are only about 83%, 47%, 37%, 40%, 77%, 61%, and 58% for seven workloads.

Second, the percentages of correct hit ratios to the ERB are not very high ranging about 40% to 80% in large configuration. This is due to the fact that a significant amount of the loads cannot be triggered early. Since those load addresses are likely unchanged across different execution instances, the correct address prediction for those loads is very high as shown in Figure 3.9. In this figure, each bar shows the

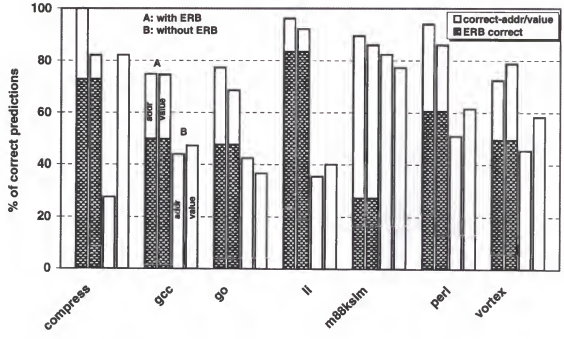


Figure 3.7. The accuracy of early resolution of loads for small configuration

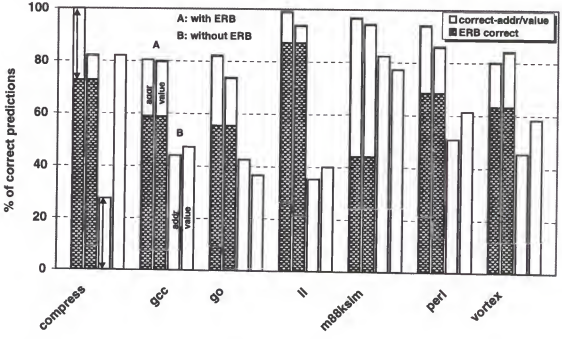


Figure 3.8. The accuracy of early resolution of loads for large configuration

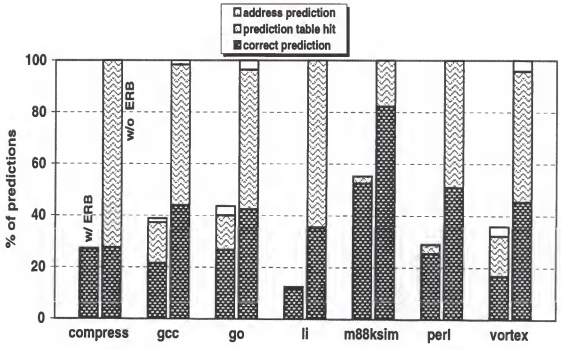


Figure 3.9. Load address prediction with/without early triggers

ratios of the total address prediction, the prediction table hits, and the correct address prediction with/without the early resolution scheme. Under the early-triggering scheme, only those loads that miss the ERB will access the prediction table. Both schemes show high hit ratios to the prediction table. However, the correct address prediction rates are much lower in stand-alone address predictor suggesting that the load addresses are usually changed across different execution instances. The correct prediction ratios with the ERB are improved for all the workloads, especially for *compress, li, m88ksim,* and *perl.*

Figure 3.10 compares the branch resolution accuracy with/without the early-triggering mechanism. Again, the early resolution of branches can achieve a significantly higher accuracy than that of the hybrid prediction scheme. Although the ERB can capture only about 60% to 90% of branches, it is able to resolve a significant

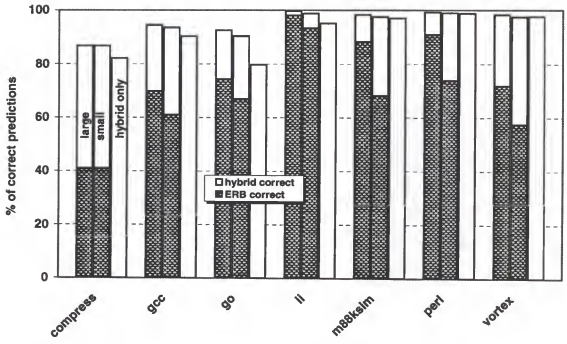


Figure 3.10. The accuracy of early resolution of branches

amount of incorrectly-predicted branches. As a result, about 5%, 5%, 13%, 4%, 1%, 1%, and 1% improvement in absolute ratios can be observed for seven workloads with large ERB. For branches, the differences of the overall prediction accuracy between large and small configurations are very minor. This is due to the fact that the additional misses with small ERB can be predicted mostly by the hybrid prediction scheme.

Early-triggering of the branch/load may generate extra instruction execution because some of the early-triggered branches/loads may not have a chance to be used, and some of the branches/loads can be mis-triggered with incorrect operands. Figure 3.11 shows the amount of loads generated by the early-triggering mechanism. There are three conditions where a load is fetched in the program order. First, the load is already triggered correctly with the result in the ERB. Second, the load is

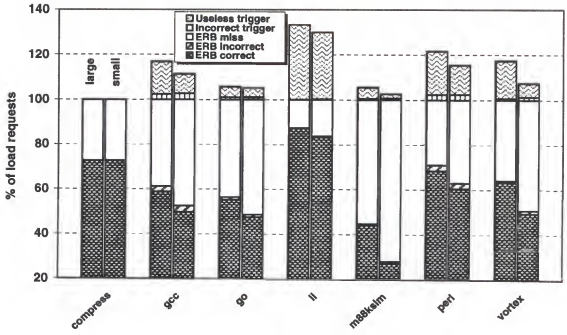


Figure 3.11. The extra loads due to early triggers

triggered incorrectly with a mismatched operand in the ERB. In this case, the load must be executed again. Third, the load misses the ERB and must be issued and executed. The incorrectly triggered loads are executed twice as shown in the figure. In addition, there are certain amount of early loads that are either never encountered in the normal execution or replaced before they are fetched in the order of the program. This category of loads is classified as the useless trigger in the figure.

There are several reasons why early-triggering mechanism exhibits relatively large amount of extra traffics which can be a negative impact of the proposed architecture. First, the data flow links from the DLT will trigger all the dependent loads/branches from the previously learned paths. Among the early-triggered instructions, only the instructions in a specific path are consumed by the normal program execution and the remaining instructions in the different paths are not used until

they are being replaced. Figure 3.12 shows an example where the case mentioned above can happen. In the figure, the links from two paths, *path_0* and *path_1*, are built by an instruction *"add $16, $9, $10"* and all the dependent links will be early triggered causing the early-triggered instructions of one path not to be used. This situation occurs mostly in loads. It seems difficult to resolve this path problem in a clean way, but a possible solution is to add a very small early-triggered load buffer with 8 to 16 entries. The small load buffer will cover most of references in multiple paths because of program locality.

Second, the processor register convention across procedure calls contributes the useless links. For example, if a register value must be preserved across the procedure call, it is possible that the callee produces spill code for the registers. This convention can create two links for the consumer, i.e., one from the caller and the other from the callee. This situation is illustrated in Figure 3.13. In the figure, the link between *func_1* to *func_0* is created uselessly because of spill code. In fact, the link within a function *func_0* itself is sufficient. A straightforward solution of code spilling problem is not to build a link if the scope of a producer and a consumer is not the same for the registers whose values need to be preserved across procedure calls.

Third, since two-source-operand instruction can have two producer links, two early triggers for a consumer instruction are generated before they are actually being used. Two-source-operand problem occurs mostly in branch instructions in our simulation model. To cure the problem, we only need to build one link from the most-recent producer. A FIFO queue, with the size equivalent to the number of

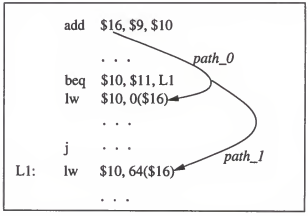


Figure 3.12. Extra traffic due to multiple paths

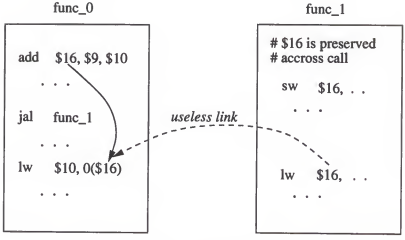


Figure 3.13. Extra traffic due to spill code

registers in a processor, is only necessary to identify relative time order of two source operands. From the relative time-stamp, only one producer can have a link to the consumer.

3.7 Summary

A new microarchitecture is proposed in this chapter to overcome the limited ILP (Instruction Level Parallelism) which can cause diminishing performance result in the multiple-issue processors. The direction to overcome the limited ILP in this

study is pursued by alleviating branch and load hazards because they are the major sources of the limitation. To be effective, prediction-based speculative execution of instruction demands very high prediction accuracies for both branches and loads. The techniques of early-resolution for branch and load outcomes are presented. The idea of early-resolution is realized by building data-flow link between the the producer and consumer instructions. Throughout the simulation, the early-resolution of branch and load instructions with data-flow link provides prediction accuracy far beyond the existing prediction methods. Several implementation issues are discussed to fully utilize the idea of the early-resolution of branches and loads. The timing of early-triggering needs special attention among the implementation issues.

CHAPTER 4
LINK-BASED HYBRID LOAD-ADDRESS PREDICTOR

4.1 Introduction

A big picture of the early-resolution of branches and loads in the previous chapter can be narrowed to a specific domain such as load address prediction. Because predicting load-address and speculating the load instruction early in the processor pipeline is a way of reducing the load latency problem and improving IPC.

Load-address prediction technique started with a simple pattern analysis study. Stride-based address predictor [10] observes that next address can be predicted correctly if the difference between two previous addresses is a constant in a static load instruction. Stride-based predictor is very effective for scientific programs in which regular memory accesses such as arrays are dominant. Since memory access patterns of integer-intensive programs are not as regular as scientific programs, context-based predictor [31] is introduced to learn non-uniform patterns and attempts to predict addresses by the patterns seen in the past. The prediction accuracies by these predictors for integer-intensive programs, however, are still very low. More recently a hybrid scheme [4], which combines both stride-based and context-based predictors, was proposed not only to achieve better address prediction accuracy but also to reduce mis-prediction overhead. In spite of the effort in the hybrid scheme, about

one-third of load addresses is not correctly predicted due to the randomness of the load address patterns.

Because the early-resolution technique which is discussed in Chapter 3 does not rely on any pattern of the previous addresses, the technique can be easily applied to capture those unpredictable addresses by the conventional predictors. The rational of link-based predictor, which adopts the early-resolution technique in Chapter 3, is explained and validated throughout the simulation in this chapter.

4.2 Load-Address Prediction Schemes

Stride-based predictor [10] provides a prediction address if the addresses in a static load instruction have a constant stride. For example, a sequence of load addresses seen in the past is *100-132-164*, it is reasonable to guess that the next address is *164(last address) + 32(stride)*. Last-address predictor can be considered as a special case of stride 0. Stride-based predictor works well for scientific programs in which regular memory accesses are dominant.

Context-base predictors [31, 31], in an attempt to adapt irregular load address patterns, match recent value history or context with the previous value histories and predict values based on the values learned previously. For example, if a sequence of values learned in the past is *100-117-121-127-111-100-117-121-127*, which entails no stride, then the value *127* is predicted in a context history of *100-117-121* because the sequence is shown before. Figure 4.1 illustrates a two-level context-based predictor: the first level table memorizes context seen in the past and the second level table maintains expected value right after a particular context. Compressing the necessary

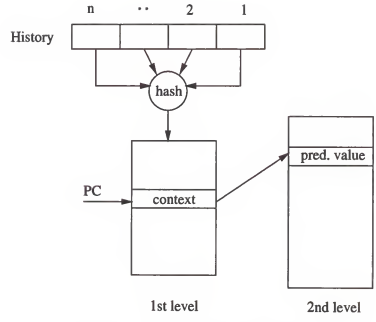


Figure 4.1. Context-based predictor

information of the context is usually made to record the context in first level history table by applying a hash function. The second level table maintains value which can be indexed by the first level context table. A hysteresis mechanism is employed in the second level table for both certainty of prediction and replacement of an entry.

The idea of hybrid predictor originates from a study by McFarling [25] which combines two or more predictor components and selects one component dynamically depending on accuracy of each prediction instance. General concept of hybrid predictor is to fully exploiting each component predictor since each component predictor works best in a certain region of the program execution.

Correlated hybrid load-address predictor (CAP), which combines stride-based and context-based predictors, is recently proposed by Beckerman et al. [4]. The CAP is proposed by the observation that load base addresses entail certain patterns in integer programs such as pointer chasing and tree traversal for which last address

predictor or stride predictor is hard to predict load address. The CAP still adopts stride-based predictor as a base component because stride-based predictor is cost-effective especially for scientific programs and some of integer programs. The CAP chooses a prediction address by either stride-based or context-based component, and the actual speculation of a load instruction is determined by its confidence mechanism.

Beckerman et al. [4] observed that a large number of unpredictable load addresses by stride-based predictor has a repeating pattern and exhibits a certain global correlation out of static load instructions. For example, there exists a repeating pattern in the link-list data structure for which conventional address predictors are known to be very inaccurate, and a global load address sequences can be seen as only constant-offset difference while the base addresses show a certain pattern. Another effort they considered in the load-address prediction is to avoid possible mis-predictions by additional confidence mechanism. So, they rebuild context-based predictor to reflect the observations they found and use the idea of hybrid predictor [25] resulting a significant of improvement of load address prediction compare to the previously-proposed address prediction schemes.

Even though they show that a 67% of all loads can be predicted while maintaining the miss prediction rate close to 1%, the remaining 33% of load references is still cannot be captured by the CAP. Those unpredictable load addresses are inherently so random that the CAP avoids prediction to reduce overhead of mis-prediction. The randomness may be occasional meaning that it can be seen as a glitch in the

previously studied pattern of load base address, hence forcing the predictor not to speculate. The glitch may even corrupt the previously studied pattern. The randomness can also be uniform in a static load instruction, in other words, the load base address pattern is so random in nature that past history of base address is hard to construct.

4.3 Link-based Load Address Prediction

For about one-third of unpredictable load references by the CAP, our link-based technique with early resolution of load address can be able to produce highly accurate prediction address. The reason is that the link-based predictor provides prediction address for load reference using *current* information instead of *past history* information of load base as in the CAP. So, regardless of randomness of load address patterns, a highly accurate address can be predicted once a link between a register update and a load has been established. The distance from base updater and load instruction, however, is a major factor which will affect the overall performance improvement by the link-based scheme.

Figure 4.2 illustrates with an example how link-based scheme can accurately predict load address while other schemes cannot easily do. Figure 4.2 contains a function *mrglist*, which is from *go* in SPECint95, and its corresponding MIPS assembler code. At the beginning of the function, *mrglist()* checks whether **list2* is EOL by executing line *(7)* of the assembler code. The line *(7)* of the assembler code is a load reference whose base address in a register *$5* is set by the caller of the function *mrglist()*. Predicting load address at line *(7)* by either stride-based or context-based

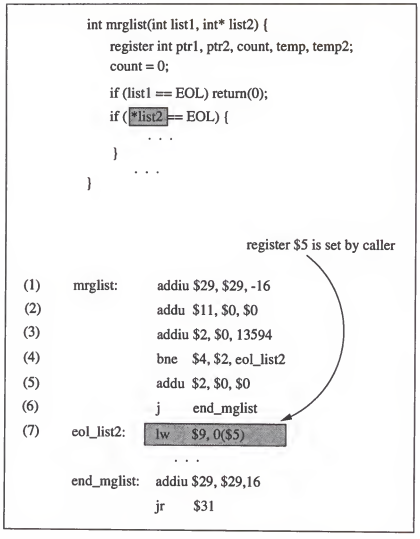


Figure 4.2. An example for which link-based scheme predicts accurately

scheme is not successful because a linked list *list2* is dynamically allocated entity and the list is now being started to be probed. Context-based scheme may be able to predict correctly for the *subsequent* load addresses from this point on. In other words, because line *(7)* is the starting point of a context a few more following load addresses are needed to construct a pattern and predict load address afterwards if the pattern was seen before. The link-based predictor, however, does not rely on either previous history or pattern. As shown in Figure 4.2, the link-based predictor

```
if an instruction is load
   look up predictor table
   if the CAP is confident
      select either stride-based or context-base prediction address
   else
      use link-based prediction address if link exists
      update DLT

for any register update instruction
   update RUT
   if a link exists
      update base value in predictor table indexed by DLT
```

Figure 4.3. Link-based predictor algorithm

can predict accurately using the base address update instruction of *(7)* in the caller site.

The CAP consists of two component predictors, i.e., stride-based predictor and context-based predictor. The unshaded portion of Figure 4.4 shows the CAP structure, and shaded-region is the extended structure to accommodate the link-based prediction into the CAP. The CAP produces both stride-based and context-based prediction addresses for every load reference, but the actual speculative load execution is decided by confidence mechanisms. The confidence mechanisms in the CAP use both saturation counters for each of the predictor component and control flow information. The control flow information keep track of global branch history when the load speculation is turned to be incorrect.

To capture a large portion of uncertain load prediction addresses by the CAP, the link-based scheme is augmented onto the CAP because the link-based scheme can provide highly accurate prediction addresses with regardless of the randomness

of load address pattern. This is possible because early-resolution based technique provides prediction address in reference to the *current* program flow of base address information instead of past history of base addresses.

Only a small budget of additional hardware structure is required to build up newly proposed link-based scheme onto the CAP as shown in Figure 4.4. The RUT (Register Update Table) and the DLT (Data-flow Link Table) need to keep track of those load references for which the CAP avoids prediction due to low confidence. A new *base* field is augmented in the CAP to record the *current* base value as early as possible.

The link-based predictor is dependent on the CAP operations, and a skeleton algorithm of the link-based predictor is summarized in Figure 4.3. More detail operation steps with an example link *"addiu $5,$28,-31404 ... lw $9,0($5)"* are explained using Figure 4.4 as follows (the number inside bracket in Figure 4.4 indicates each step).

step (1) A prediction of low confidence load address is generated by the CAP.

step (2) Build a link between the load-base-update instruction and the load instruction into the DLT in reference to the RUT.

step (3) From next time, lookup the DLT if the base-update instruction, *"addiu $5,$28,-31404"* in this example, is encountered during the program execution.

step (4) The base value in the link-based predictor is updated through the DLT. When a miss occurs in this step, no replacement activity in the predictor component is performed to avoid any side-effect by the link structure.

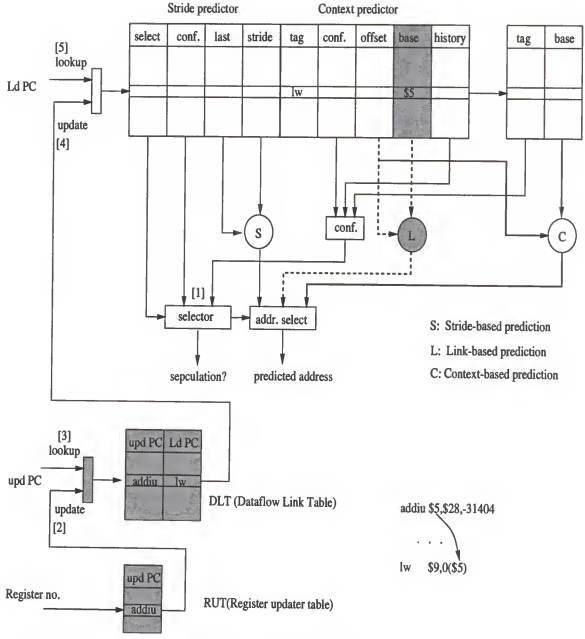


Figure 4.4. The link-based scheme on the CAP(Correlated Address Predictor)

step (5) Soon after when a load instruction comes, *"lw $9,0($5)"* in this example, both a predicted address from the CAP and a predicted address from the link-based scheme are prepared. The predicted address by the link-based component is used only when the CAP informs not to speculate.

4.4 Link-based Predictor in Processor Pipelining

The benefit of the link-based prediction can be reduced if the distance between the register updater and the load is very short. Therefore, careful design details need be considered to overcome the distance problem. The *Simplescalar* [6] pipeline is used as a base model to illustrate and validate the idea of link-base address prediction scheme in this section.

The Simplescalar pipeline closely follows MIPS [19] architecture and it is modeled to perform out-of-order execution. The detailed explanation of each pipeline stage can be found by Burger and Austin [6]. In short, its pipelining is consists of five stages, i.e., IF (Instruction Fetch), DA (Decoding and Scheduling), EX (Execution), WB (Write-back), and CT (Commit). Some of the queue structures such as *ifq* (instruction fetch queue), *readyq* (ready queue), *earlyq* (early queue), and *eventq* (event queue) are shown in Figure 4.5 along with major functional units. Five-stage Simplescalar pipeline goes through six cycles for load and store instructions, in other words, the Simplescalar pipeline can be viewed as performing two consecutive execution stages for memory references, i.e., address generation and memory access.

Integration of the link-based predictor with the base model pipeline is constructed as Figure 4.5. A predicted load address is ready at IF stage after lookup

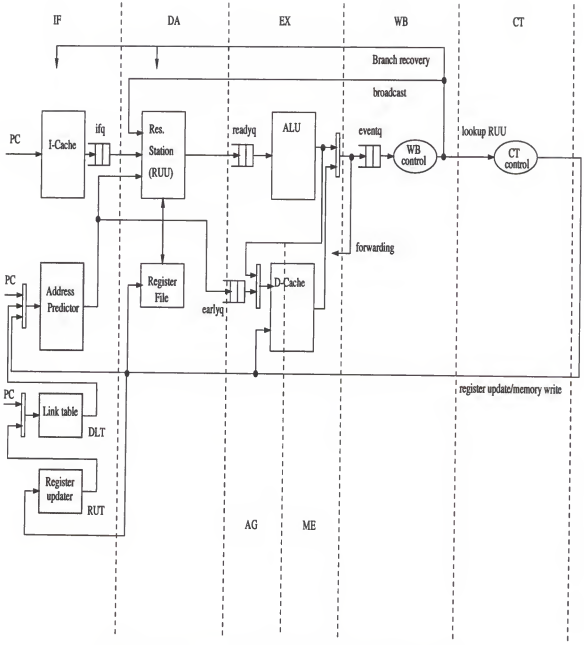


Figure 4.5. Link-based prediction under pipeline stages

of the prediction table is finished, and speculative cache access is initiated at DA cycle. The predictor component's fields are updated if any load instruction reaches CT stage even though the prediction result can be known earlier than CT stage. The forwarding paths from earlier pipeline stage to the predictor structures are also considered to overcome the distance problem. The RUT, the DLT, and corresponding predictor table fields, if exists, are also updated whenever any register update instruction enters CT stage.

A challenging part of the link-based prediction is how effectively deal with the situation when the distance between base register update and load is very short. A conservative approach for the link-base predictor to solve the distance problem is simply to discard those short distance loads seen by the data-flow link. If this approach is applied to the Simplescalar pipeline, the link-based predictor will predict only if the number of cycle differences between the the base updater and the load is greater than *one*. Figure 4.6 assumes single-issue pipeline and shows the conservative pipeline execution under a set of instruction sequences. Note that *lw(AG)*/lw at cycle 4 means both decoding of normal load and address generation of speculative cache access are performed at the same time. In the figure, if the cycle distance is one the link-based predictor does not reduce load latency.

Figure 4.7 shows aggressive approach under a set of instruction sequences. The example sequence of the instructions in upper part of Figure 4.7 has distance two, and it is possible for the aggressive approach to take advantage of the link fully. For example, a prediction address can be ready at clock 3 using the value forwarded from

cycle distance 3

cycle	IF	DA	EX	WB	CT	
1	add					
2		add				add $8, $8, $9
3			add			nop
4	*lw(AG)* / lw			add		nop
5		lw	***lw(ME)***		add	lw $10, 0($8)
6			lw(AG)	***lw***		
7			lw(ME)		***lw***	***lw*** : **speculation**
8				lw		
9					lw	

cycle distance 2

cycle	IF	DA	EX	WB	CT	
1	add					
2		add				add $8, $8, $9
3	*lw(AG)* / lw		add			nop
4	*lw(AG)*	lw		add		lw $10, 0($8)
5			*lw(ME)* / lw(AG)		add	
6			lw(ME)	***lw***		
7				lw	***lw***	***lw*** : **speculation**
8					lw	

Figure 4.6. Conservative link-based prediction examples under pipeline

add instruction at EX stage, and a speculative cache access, indicated by *lw(ME)*, can start at clock cycle 4. A speculative cache access by address prediction is initiated two-cycle ahead of the normal load instruction suggesting that further speculation of load-dependent instruction is also possible.

Even when the cycle difference is only one, the aggressive approach can exploit benefit. It can be seen easily that there is no chance for the link-based scheme to

cycle distance 2

cycle	IF	DA	EX	WB	CT	
1	add					
2		add				add $8, $8, $9
3	*lw(AG)* / lw		add			nop
4		lw	*lw(ME)*	add		lw $10, 0($8)
5			lw(AG)	*lw*	add	
6			lw(ME)		*lw*	*lw* : speculation
7				lw		
8					lw	

cycle distance 1

cycle	IF	DA	EX	WB	CT	
1	add					
2	*lw(AG)* / lw	add				add $8, $8, $9
3	*lw(AG)*	lw	add			lw $10, 0($8)
4			*lw(ME)* / lw(AG)	add		
5			lw(ME)	*lw*	add	
6				lw	*lw*	*lw* : speculation
7					lw	

Figure 4.7. Aggressive link-based prediction examples under pipeline

use the very previous instruction's result at IF stage. But, if the speculative cache access can be initiated at address generation stage of a normal load, the load data can be ready at one cycle ahead of the normal load instruction. Like the conservative approach, data forwarding from the load base updater is also assumed. Lower part of Figure 4.7 illustrates this approach when the two instructions are one cycle apart. At cycle 3, the result of *add* instruction is forwarded into the following load, and the

normal address calculation and the speculative cache access, indicated by *lw/lw_sp*, are performed at cycle 4 simultaneously. In fact, the speculative cache access at cycle 3 should not be issued in reference to dependency chain at DA stage. On the other hand, if the update instruction is also a load such as a sequence of *"lw $10, 0($11); lw $13, 0($10)"* where the second load uses the result of the first load, we cannot utilize the benefit of the aggressive scheme unless the first load itself is performed speculatively.

The recovery activities are performed at WB stage if the predicted address is flagged as incorrect at address generation stage. The recovery activities involve issuing normal cache access and rescheduling any load-dependent instruction that is already issued using the speculated load result.

4.5 Performance Evaluation

4.5.1 Simulation Model

The base structure of simulation model in Chapter 3 is also used here. The DLT and the RUT are the same structure as before but the ERB is merged into the CAP. In fact, a minimal information of the ERB in Chapter 3 is put into the CAP, i.e., *base* register value field. No other information are needed because the CAP maintains the remaining information.

A fast functional simulator in the *Simplescalar* tool is used at the front-end simulation environment, and our back-end link-based predictor consumes the information provided from the front-end simulator. As the back-end simulator interprets each instruction, the necessary steps such as updating the RUT, building the data-flow

links to the DLT, and updating of link-based predictor for the load instructions are performed. In case that the instruction is a load, a matching entry in the link-based predictor must further decide which prediction component is actually used. The prediction by the data-flow link is active only when the CAP informs not to speculate. This also implies that the data-flow link between a load instruction and its base register update instruction(s) is established only for those non-confident loads by the CAP. The CAP updates its table information whenever the real load address is known with regardless of speculation request.

Seven SPECint95 benchmark programs are studied through this simulation. To asses the performance of the link-based hybrid predictor easily, almost the identical simulation parameters as Beckerman et al. [4] are used throughout this simulation. The simulation parameters are as follows: the predictor is 4K-entry 2-way set-associative, confidence fields of stride-based and context-based use 2-bit saturation counters, control flow information consist of 8-bit of global branch history register, selector also uses 2-bit saturation counter, and load base history length is 3. A few sensitivity studies are performed by changing the DLT and the RUT stack sizes, i.e., the DLT is 32/64-entry 8-way set associative and 0/16-entry RUT stack is constructed in this simulation model. Variable-distance simulation is modeled to reflect the cycle distance in the functional simulator. Setting of distance 1 implies that the additional predictions are made by the link-based scheme without any distance constraint between register updater and load instruction, on the other hand, setting of distance 9 allows to predict when register updater and load instruction are

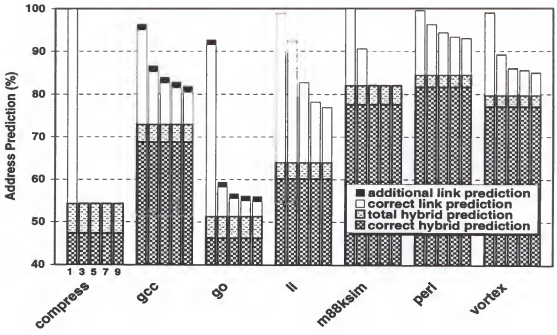


Figure 4.8. Link-based address prediction accuracy with 32-set DLT

9 or more instruction apart during the execution of the program. Like Beckerman et al. [4], 30-million instruction references are traced and statistics are collected without warm up.

4.5.2 Evaluation Result and Discussion

Figure 4.8 and Figure 4.9 show prediction accuracies with 32-set and 64-set DLTs respectively for seven SPECint95 programs.

The additional predictions by the link-based predictor and their accuracies from cycle distance 1 to 9 are stacked up on the baseline CAP. The additional hardware structure of the link-based predictor does not affect the prediction accuracies of the original CAP. In other words, the prediction accuracies by the CAP component in the figures are the same as the accuracies of baseline CAP alone because any update in predictor entry is relying on the update policy of the baseline CAP.

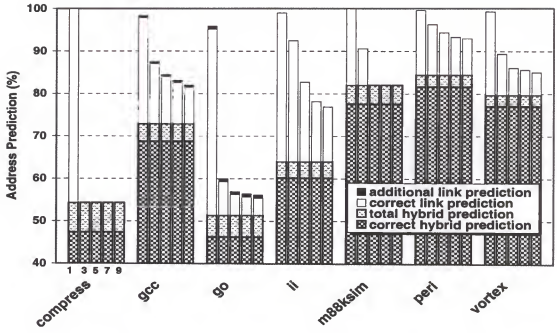
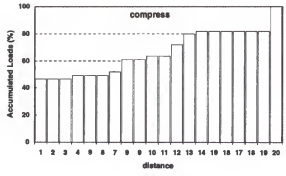
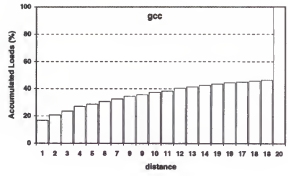
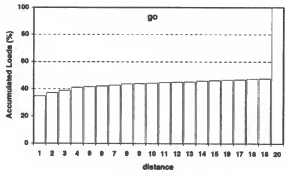


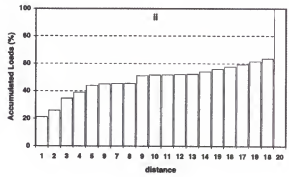
Figure 4.9. Link-based address prediction accuracy with 64-set DLT

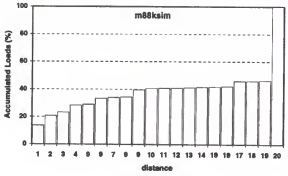
As shown in Figure 4.8 and Figure 4.9, prediction accuracies of the baseline CAP are very high even though the accuracies are not as high as the published paper in Beckerman et al. [4] which models IA-32 instruction set architecture, the simulation model in this study, on the other hand, follows MIPS instruction set architecture. The conservative speculation by confidence mechanism in the baseline CAP can avoid a lot of possibly incorrect predictions and reduce recovery overhead caused by incorrect speculation. Several observations can be found from the simulation results. First, a large number of load addresses is not predictable by the CAP because of randomness of the load addresses in the SPECint95. But, the link-based scheme can capture those unpredictable addresses very accurately. Benchmarks *compress*, *go*, and *li* are the cases where the random behavior of load references are exhibited significantly in the SPECint95. The overall prediction ratios of those three programs by the CAP

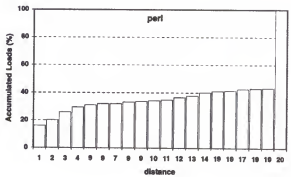












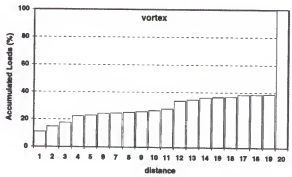


Figure 4.10. Cycle distance distributions of the SPECint95

are 54.3%, 51.4%, and 63.9% respectively. Additional prediction improvement by the link-based prediction for those three programs is significant especially when the value of distance constraint is small. For example, in *li*, the additional prediction rates out of total loads from cycle distance 1 to 9 in 64-set DLT are 35.2%, 29.2%, 21.7%, 18.1%, and 17.7% respectively. The accuracies by the link-based prediction are even higher than the CAP. For example, in benchmark *li*, the accuracy by the link-based component only is close to 99% .

Secondly, most of the unpredictable addresses by the CAP are very close to the base-update instructions in terms of cycle distance (Figure 4.10 shows the distribution of cycle distance for each of the SPECint95). A sharp decrease of prediction rate by the link-based scheme can back up this observation. For example, in *compress* and *go*, two facts—a significant drop of predictions by the link-based scheme and almost flat shape of distance when the distances are very short—indicate a large portion of unpredictable load instructions is very close to the update instructions. Benchmark *li* is one of the exceptions in which the prediction rate by the link-based scheme is decreasing smoothly.

Lastly, very small hardware resources are needed to capture unpredictable addresses. No significant increment in prediction rate can be seen as the number of DLT set is doubled. From Figure 4.8 and Figure 4.9, the prediction rates are almost identical for 32-set and 64-set DLT except benchmark *go*. In *go*, prediction rate increases from 40.1% to 43.8% when the DLT size is doubled. Very high prediction accuracies by the link-based scheme without using the RUT stack in the figures imply control

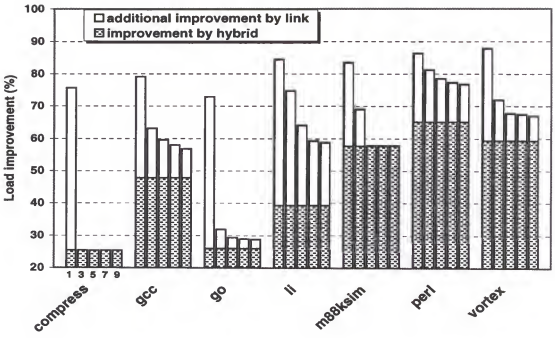


Figure 4.11. Load improvement by address prediction with 32-set DLT

and recursive structure in the SPECint95 are somewhat regular that the CAP can predict load addresses accurately for which control correlation exhibits.

The expected performance comparison between the link-based predictor and the CAP is computed in a simplified fashion by considering the number of stall cycles in loads. The charged cycles for each load are as follows: one cycle stall for not predicted load, zero cycle stall for correctly predicted load, and two stall cycles for incorrectly predicted load. The checker-board regions in Figure 4.11 and Figure 4.11 show load improvement by the CAP, and the white regions show additional improvement by the link-based scheme over no address prediction. Only slight performance differences in benchmarks *gcc* and *go* can be seen between two figures. The additional load improvement by the link-base scheme is significant especially when the value of distance constraint is small. The additional improvement of 64-set DLT for the

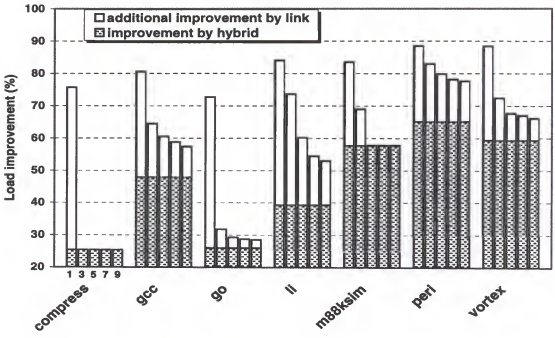


Figure 4.12. Load improvement by address prediction with 64-set DLT

SPECint95, on the average, is 36.2%, 14.2%, 8.7%, 7.2%, and 6.5% when the value of distance constraint is 1, 3, 5, 7, and 9 respectively.

4.6 Summary

A different direction of load address prediction scheme is proposed in this chapter to reduce latency of load-dependent instruction. The new scheme attempts to predict the addresses from the most recent data-flow information between load base updater and load instruction. Using data-flow information for address prediction is not so pattern-dependent as most conventional predictors that the proposed link-based scheme can predict addresses very accurately. A very small hardware structure is augmented to employ the link-based predictor onto the context-based hybrid address predictor (CAP). Several observations are found throughout the simulation. First, even though highly accurate load speculations are possible by the CAP, a large

number of load references is still cannot be predicted accurately by either stride-based or context-based predictor due to the randomness of load addresses in the integer programs. A majority of the remaining addresses is very accurately predictable by the link-based predictor. Second, a large portion of unpredictable addresses by the CAP is very close to the base-update instructions in terms of cycle distance. This fact may reduce the effectiveness of the link-based predictor. Third, very small hardware budget is required for the link-based scheme to capture the remaining pattern-less load addresses.

CHAPTER 5
CONCLUSION

Throughout the observations and performance evaluation results on multiple-access caches, newly proposed schemes in this dissertation can be served as a design framework of multiple-access caches in memory hierarchy by the following reasons. First, a set of more generalized rehash functions shows dramatic improvement over the traditional rehash function of flipping only the highest-order index bit, especially, in scientific programs. Second, the existing multiple-access caches require additional information in the form of rehash bits to provide more accurate search and replacement for the requested cache line. The simulation results of the proposed search and replacement algorithm show that the search according to the MRU/LRU sequence in 2-way column associative caches can achieve better performance than the rehash-bit in terms of both overall and primary hit ratios. Extending the proposed scheme into 4-way multiple-access cache, the performance using the propose approach is viable to index-vector scheme with much less cost and complexity. Finally, a newly proposed indirect cache access mechanism seems to overcome the criticism of swapping problem in multiple access caches because swapping only tags without swapping the corresponding data in the cache can achieve a higher hit ratio to the primary location.

A small bit array is added for the indirect access, and the timing simulation confirms that the indirect path does not lengthen the overall cache access time.

Due to the inherent limitations of control and data speculation, increasingly aggressive implementations of conventional wide-issue superscalar processor yield only diminishing returns in IPC. To jump over the barrier of current branch and load address/value prediction accuracies, early resolution mechanism of branch and load instructions shows uniqueness of its ability to identify the critical instructions that cannot be handled effectively by speculation. The evaluation results show that by selective early triggering and speculative instruction execution, the newly proposed scheme shows a great promise for achieving the next level of performance without requiring undue hardware resources. In additional, the idea of the proposed early-resolution technique is applied to the most current address predictors to further improve address prediction accuracy. The simulation results of the proposed enhancement technique show that only a small augmentation of hardware structure can capture, very accurately, most of the unpredictable load addresses by the conventional predictors. In spite of distance constraint, the proposed early-resolution technique can provide a new dimension in the design of wide-issue speculative processors.

REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming," *ACM Trans. on Computer Systems*, Vol. 6(4), Nov. 1988, pp. 393–431.

[2] A. Agarwal and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," *Proc. 20th Int'l Symp. on Computer Architecture*, San Diego CA, May 1993, pp. 179–190.

[3] T. Austin, D. Pnevmatikatos, and G. Sohi, "Streamlining Data Cache Access with Fast Address Calculation," *22nd Annual Int'l Symp. on Computer Architecture*, S. Margherita Ligure Italy, June 1995, pp. 369–380.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *26th Annual Int'l Symp. on Computer Architecture*, Atlanta GA, May 1999, pp. 54–63.

[5] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. Shen, "Load Execution Latency Reduction," *ACM 1998 Int'l Conf. on Supercomputing*, Melbourne Australia, Aug. 1998, pp. 29–36.

[6] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, CS Department, University of Wisconsin-Madison, June 1997.

[7] B. Calder, D. Grunwald, and J. Emer, "Predictive Sequential Associative Cache," *Proc. 2nd Symp. on High-Performance Computer Architecture*, San Jose CA, Jan. 1996, pp. 244–253.

[8] J. Dennis and D. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Proc. of 2nd Int'l Symp. on Computer Architecture*, Houston TX, June 1975, pp. 125–131.

[9] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, Vol. 17(5), Sep. 1997, pp. 12–19.

[10] R. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit For Pipelined Processors," *IBM Journal of Research and Development*, Vol. 37(4), pp. 547–564, July 1993.

[11] M. Evers, S. Patel, R. Chappel, and Y. Patt, "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," *25th Annual Int'l Symp. on Computer Architecture*, Barcelona Spain, June 1998, pp. 52-61.

[12] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes," *31st Int'l Symp. on Microarchitecture*, Dallas TX, Dec. 1998, pp. 59-68.

[13] A. Gonzalez, M. Valero, N. Topham, and J.M. Parcerisa, "Eliminating Cache Conflict Misses Through XOR-Based Placement Functions," *Proc. 11th Int'l Conference Supercomputing*, Vienna Austria, 1997, pp. 76–83.

[14] J. Gonzalez and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," *Proc. 11th Int'l Conf. on Supercomputing*, Vienna Austria, 1997, pp. 196–203.

[15] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *IEEE Computer*, Vol. 30(9), Sep. 1997, pp. 79–85.

[16] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, 2nd-Edition, Morgan-Kaufmann, San Francisco CA, 1996.

[17] M. Hill "A Case for Direct-Mapped Caches," *IEEE Computer*, Vol. 21(12), Dec. 1988, pp. 25–40.

[18] K. Hua, A, Hunt, L. Liu, J-K. Peir, D. Pruett, and J. Temple, "Early Resolution of Address Translation in Cache Design," *1990 Int'l Conf. on Computer Design*, Boston MA, Sep. 1990, pp. 408–412.

[19] G. Kane and J. Heinrich, *MIPS RISC Architecture,* Prentice Hall, Englewood Cliffs NJ, 1992.

[20] R. Kessler, R. Jooss, A Lebeck, and M. Hill, "Inexpensive Implementation of Set-Associativity," *Proc. 16th Int'l Symp. on Computer Architecture*, Honolulu HI, May 1988, pp. 131–139.

[21] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction,", *Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge MA, Oct. 1996, pp. 138–147.

[22] M. Lipasti and J. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. of the 29th Annual ACM/IEEE Int'l Symp. on Microarchitecture*, Paris France, Dec. 1996, pp. 226–237.

[23] M. Lipasti and J. Shen, "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE Computer*, Vol. 30(9), Sep. 1997, pp. 59–66.

[24] W. Lynch, G. Lauterbach, and J. Chamdani, "Low Load Latency through Sum-Addressed Memory (SAM)," *25th Annual Int'l Symp. on Computer Architecture*, Barcelona Spain, June 1998, pp. 369–379.

[25] S. McFarling, "Combining Branch Predictors," DEC WRL Technical Report TN-36, Palo Alto CA, June 1993.

[26] S. Pan, K. So, and J. Rameh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. of 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston MA, Oct. 1992, pp. 76–84.

[27] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computer*, Vol. 30(9), Sep. 1997, pp. 51–57.

[28] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thmoas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, Vol. 17(2), Mar/Apr 1997, pp. 34–44.

[29] J. Peir, W. Hsu, H. Young, and S. Ong, "Improving Cache Performance with Balanced Tag and Data Paths," *Proc. 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge MA, Oct. 1996, pp. 268–278.

[30] A. Roth, A. Moshovos, and G. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proc. of the 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose CA, Oct. 1998, pp. 115–126.

[31] Y. Sazeides and J. Smith, "The Predictability of Data Values," *MICRO 30 Proceedings*, Triangle Park NC, 1997, pp. 248–258.

[32] A. Seznec, "A Case for Two-way Skewed-Associative Caches," *Proc. 20th Int'l Symp. on Computer Architecture*, San Diego CA, May 1993, pp. 169–178.

[33] A. Smith, "Cache Memories," *Computing Surveys*, Vol. 14(4), Sep. 1982, pp. 473–530.

[34] J. Smith, "Decoupled Access/Execute Computer Architecture," *Proc. of 9th Int'l Symp. on Computer Architecture*, May 1982, pp. 231–238.

[35] J. Smith and S. Vajapeyam, "Trace Processors: Moving to Fourth-Generation Microarchitectures," *IEEE Computer*, Vol. 30(9), Sep. 1997, pp. 68–74.

[36] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. of 22nd Int'l Symp. on Computer Architecture*, S. Margherita Ligure, Italy, May 1995, pp. 414–425.

[37] SPEC CPU95 Benchmark Suite, Version 1.10, SPEC Corp., Manassas VA, Aug. 1995.

[38] H.S. Stone, *High-Performance Computer Architecture*, 2nd Ed., Addison-Wesley, Reading MA, 1990.

[39] Sun Microsystems, "Introduction to Shade," v5.32c, Mountain View CA, Sun Microsystems, 1997.

[40] S. Wilton and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," DEC WRL Research Report 93/5, Palo Alto CA, July 1994.

[41] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Vol. 16(2), April 1996, pp. 28–40.

[42] T. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *19th Annual Int'l Symp. on Computer Architecture*, Queensland Australia, May 1992, pp. 124–134.

[43] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History," *20th Annual Int'l Symp. on Computer Architecture*, San Diego CA, May 1993, pp. 257–266.

[44] C. Zhang, X. Zhang, and Y. Yan, "Two Fast and High-Associativity Cache Schemes," *IEEE Micro*, Vol. 17(5), Sep/Oct 1997, pp. 40–49.

BIOGRAPHICAL SKETCH

Byung-Kwon Chung received his B.S. degree in electronic engineering from Sogang University in Seoul Korea in 1982. After military service, he joined Gold-Star Co., Ltd, currently LG Electronics, where he conducted research and development on micro-computer hardware design. He also worked as an application engineer of micro-computer development systems for Tektronix Inc., until 1989. In 1991, he received an M.S. degree from Western Illinois University in computer science. In 1999, he received a doctoral degree in the Department of Computer and Information Science and Engineering at the University of Florida. His major research areas are cache memories and processor pipelining in computer systems.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jih-Kwon Peir, Chairman
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Randy Y. Chow
Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sanguthevar Rajasekaran
Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Abdelsalam Helal
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Paul W. Chun
Professor of Biochemistry and
Molecular Biology

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1999

M. Jack Ohanian
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School